

# Parallel Graph Algorithms for Finding Weighted Matchings and Subgraphs in Computational Science

## Inauguraldissertation

zur Erlangung der Würde eines Doktors der Philosophie  
vorgelegt der Philosophisch-Naturwissenschaftlichen Fakultät  
der Universität Basel

von

**Madan Sathe**

aus Borken (Westfalen), Deutschland

Basel, 2012



Originaldokument gespeichert auf dem  
Dokumentenserver der Universität Basel: **edoc.unibas.ch**.

Dieses Werk ist unter dem Vertrag "Creative Commons  
Namensnennung–Keine kommerzielle Nutzung–Keine  
Bearbeitung 2.5 Schweiz" lizenziert. Die vollständige  
Lizenz kann unter  
<http://creativecommons.org/licences/by-nc-nd/2.5/ch>  
eingesehen werden.



*Attribution – NonCommercial – NoDerivs 2.5 Switzerland*

*You are free:*



**to Share** — to copy, distribute and transmit the work

*Under the following conditions:*



**Attribution** — You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).



**Noncommercial** — You may not use this work for commercial purposes.



**No Derivative Works** — You may not alter, transform, or build upon this work.

*With the understanding that:*

**Waiver** — Any of the above conditions can be *waived* if you get permission from the copyright holder.

**Public Domain** — Where the work or any of its elements is in the *public domain* under applicable law, that status is in no way affected by the license.

**Other Rights** — In no way are any of the following rights affected by the license:

- Your fair dealing or *fair use* rights, or other applicable copyright exceptions and limitations;
- The author's *moral* rights;
- Rights other persons may have either in the work itself or in how the work is used, such as *publicity* or privacy rights.

**Notice** — For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to the web page <http://creativecommons.org/licenses/by-nc-nd/2.5/ch>.

---

**Disclaimer** — The Commons Deed is not a license. It is simply a handy reference for understanding the Legal Code (the full license) – it is a human-readable expression of some of its key terms. Think of it as the user-friendly interface to the Legal Code beneath. This Deed itself has no legal value, and its contents do not appear in the actual license. Creative Commons is not a law firm and does not provide legal services. Distributing of, displaying of, or linking to this Commons Deed does not create an attorney-client relationship.

---

Genehmigt von der Philosophisch-Naturwissenschaftlichen Fakultät auf  
Antrag von

Prof. Dr. Helmar Burkhart  
Prof. Dr. Olaf Schenk  
Prof. Dr. Rob Bisseling

Basel, den 26.06.2012

Prof. Dr. Jörg Schibler,  
Dekan





*To my parents Manohar & Smita and my wife Angelina*



---

# Abstract

---

Graphs constitute one of the most crucial data structures in computational science and engineering. The algorithms operating on these data structures are computational kernels in various data intensive applications; for instance, in social network analysis, in computational biology, and in scientific computing. In order to enhance the computational performance of graph algorithms, techniques of high-performance computing represent the key to run these algorithms on massively parallel architectures. However, graph algorithms typically feature irregular memory access patterns and low arithmetic intensities which present a challenge for the engineering of efficient parallel graph algorithms.

In this thesis, a parallel auction-based weighted matching implementation, PAUL, is designed to solve the bipartite weighted graph matching problem on distributed memory clusters. This thesis outlines that the solving of graph matching problems can be significantly accelerated in various data intensive applications such as the graph similarity of protein-protein interaction networks and the permutation of large entries onto the main diagonal of a matrix in numerical linear algebra.

Furthermore, a dense subgraph problem is identified in parallel numerical linear algebra whose solution considerably improves the convergence and robustness of hybrid linear solvers. Three heuristics are designed and implemented to solve the  $\mathcal{NP}$ -hard combinatorial problem efficiently; the most promising one is based on evolutionary algorithms. The impact of solving the heuristics is demonstrated in the hybrid linear solver PSPIKE when solving data intensive applications in arterial fluid dynamics and PDE-constrained optimization.



---

# Acknowledgments

---

I would like to thank Prof. Dr. Helmar Burkhart and Prof. Dr. Olaf Schenk for giving me the opportunity to do my PhD at the University of Basel and for their research guidance, their support, inspirations, advice, and confidence.

I would also like to thank Prof. Dr. Rob Bisseling for kindly agreeing to act as a co-referee in the thesis committee.

I am grateful to the colleagues of our research group for fruitful and inspiring discussions: Sandra Burri, Dr. Matthias Christen, Robert Frank, Dr. Martin Guggisberg, Dr. Florian Müller, Jürg Senn, Max Rietmann, and Dr. Sven Rizzotti. Many thanks go also to Ye Zhao, David Eichenberger, and Christian Mächler who have done their bachelor and master projects under my guidance.

I wish to thank Prof. Dr. Ahmed Sameh for giving me the opportunity to stay for five weeks at his research group at Purdue University, USA.

I would like to acknowledge following colleagues who share their professional and personal experiences with me: Prof. Dr. Ananth Grama, Johannes Huber, Dr. Giorgos Kollias, Dr. Drosos Kourounis, Dr. Johannes Langguth, Prof. Dr. Fredrik Manne, Prof. Dr. Murat Manguoglu, Dr. Faisal Saied, Dr. Dominik Szczerba, Silke Wagner, Prof. Dr. Andreas Wächter, and Dr. Albert-Jan Yzelman.

Many thanks go to Angelina Asberger-Sathe, Dr. Matthias Christen, Gary Davidoff, and Sandra Kim Zerr for proofreading this thesis.

I would like to thank my wife Angelina for her love, understanding, support, endless patience, and warm encouragement when it was most required. Finally, I am forever indebted to my wonderful and lovely parents, Manohar (born in Roha, India) and Smita (born in Mumbai, India). They receive my deepest gratitude and love for their dedication. I owe them everything.

This work was supported by the CTI project no. 8582.1 ESPP-ES entitled “Computational Production Planning Methods for Automotive Press Tools” and by the Swiss National Supercomputing Centre (CSCS).

---

# Contents

---

<b>Contents</b>	<b>i</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Contributions . . . . .	9
 <b>I Graph Theory in Computational Science</b>	 <b>11</b>
<b>2 Graph Problems and Algorithms</b>	<b>13</b>
2.1 Preliminaries . . . . .	13
2.1.1 Combinatorial Graph Problems . . . . .	15
2.2 Graph Matching Problem . . . . .	18
2.2.1 Weighted Graph Matching . . . . .	18
2.2.2 Landscape of Graph Matching Algorithms . . . . .	19
2.3 Graph Partitioning and Ordering . . . . .	22
2.3.1 Graph and Hypergraph Partitioning Models . . . . .	22
2.3.2 Multilevel Framework and Software . . . . .	24
2.3.3 Spectral Orderings . . . . .	26
2.4 Dense Subgraph Problem . . . . .	27
2.4.1 Landscape of Algorithms . . . . .	27
2.4.2 Evolutionary Algorithms . . . . .	29
 <b>3 Auctions in Bipartite Graph Matching</b>	 <b>33</b>
3.1 Auction Theory . . . . .	33
3.2 Sequential Auction Algorithms . . . . .	35
3.3 Existing Parallel Auction Algorithms . . . . .	38

<b>II</b>	<b>Parallel Graph Matching</b>	<b>41</b>
<b>4</b>	<b>Design of Parallel Auction Algorithms</b>	<b>43</b>
4.1	PAUL — A Parallel Auction Algorithm Implementation . . .	43
4.1.1	Performance Aspects . . . . .	48
4.2	$\epsilon$ -Scaling Mechanisms . . . . .	49
4.2.1	Normalized Edge Weights . . . . .	50
4.2.2	$\epsilon$ -Scaling Strategies . . . . .	50
4.2.3	Optimality and Convergence . . . . .	53
<b>5</b>	<b>Software Implementation Aspects of PAUL</b>	<b>55</b>
5.1	Input Data . . . . .	55
5.2	Work Flow in PAUL . . . . .	56
<b>III</b>	<b>A Dense Subgraph Problem as Building Block in Numerical Linear Algebra</b>	<b>59</b>
<b>6</b>	<b>Design of Scalable Hybrid Linear Solvers</b>	<b>61</b>
6.1	Hybrid Linear Solvers . . . . .	61
6.2	PSPIKE — A Scalable Hybrid Linear Solver . . . . .	62
6.2.1	The SPIKE Algorithm . . . . .	63
6.2.2	The PSPIKE Algorithm and Implementation Issues . . . . .	66
6.3	Graph Problems in PSPIKE . . . . .	70
<b>7</b>	<b>Dense Subgraph Problem</b>	<b>75</b>
7.1	Quality Measures . . . . .	76
7.2	Heuristics . . . . .	78
7.2.1	FIRSTFIT . . . . .	78
7.2.2	DELETMIN . . . . .	79
7.2.3	Evolutionary Algorithms . . . . .	82
7.3	Comparison of the Heuristics . . . . .	86
<b>8</b>	<b>Software Implementation Aspects of PSPIKE</b>	<b>87</b>
8.1	Input Data . . . . .	87
8.2	The PSPIKE Phases . . . . .	88
8.3	Combining Reordering Strategies . . . . .	89



<b>IV Data Intensive Applications</b>	<b>93</b>
<b>9 Applications</b>	<b>95</b>
9.1 Graph Similarity . . . . .	95
9.2 Arterial Flow Simulation . . . . .	102
9.3 Optimal Control of Partial Differential Equations . . . . .	104
<b>10 Computational Results</b>	<b>107</b>
10.1 Experimental Testbed . . . . .	107
10.2 Benchmark Results with PSPIKE . . . . .	108
10.2.1 Florida Collection . . . . .	108
10.2.2 Arterial Flow Simulation . . . . .	114
10.2.3 Optimal Control of Partial Differential Equations . .	124
10.3 Benchmark Results with PAUL . . . . .	128
10.3.1 Sparse Linear Algebra . . . . .	130
10.3.2 Artificial Dense Bipartite Graphs . . . . .	133
10.3.3 Image Feature Matching . . . . .	134
10.3.4 Graph Similarity . . . . .	138
<b>V Conclusions &amp; Outlook</b>	<b>147</b>
<b>11 Conclusions and Outlook</b>	<b>149</b>
<b>Bibliography</b>	<b>157</b>
<b>A User Manuals</b>	<b>181</b>
A.1 PSPIKE . . . . .	181
A.1.1 Arguments of PSPIKE . . . . .	181
A.1.2 The Option File pspike.opt . . . . .	183
A.1.3 A Small Example . . . . .	187
A.2 PAUL . . . . .	189
A.2.1 A Small Example . . . . .	192
<b>List of Symbols</b>	<b>195</b>
<b>Index</b>	<b>199</b>



# Chapter 1

---

## Introduction

---

A graph is a powerful and versatile representation of data and provides an abstract view of complex data and their interactions. In real-world networks, such as social, biological, and technical networks, a large amount of data is generated and stored. Being in the petascale computing era, graphs are pervasive in modeling data intensive applications in VLSI chip layout, computational biology, data mining, numerical linear algebra, and network analysis, but processing and analyzing the data are computationally challenging [2].

For instance, the social network service Facebook with millions of users generates a huge amount of data every day. In order to get insights into functions and topological structures of the network, a social graph can be constructed from the massive dataset: people are represented by vertices and two people are connected by an edge if they are related to each other. Both constituents might be labeled with attributes or numerical values. Typically, the graphs are sparse as people know only a small number of the overall number of people. The sparse graph typically features a skewed vertex distribution, a low graph diameter, and the availability of dense subgraphs encoding communities [155, 172].

In social network analysis, graph theoretical metrics such as betweenness centrality and traversals are of growing interest [156, 169]. The computation of betweenness centrality inherently depends on the solution of the shortest path problem as vertices lying on many shortest paths are considered to be more important and have a higher betweenness than other vertices. The shortest path is a path connecting a source vertex with a destination vertex while having the minimum distance over all existing

paths.

In general, analyzing large graphs requires, on the one hand, efficient data structures [75] and parallel algorithms from graph theory, but on the other hand, advanced techniques of high performance computing [112].

In graph theory, the design and engineering of parallel graph algorithms running on a parallel random access machine (PRAM) has been researched a long time ago [123]. However, PRAM does not realistically represent current high performance computing systems as no synchronization costs, no communication costs, and no parallel overhead are assumed in the computation. Thus, there is a need to engineer parallel graph algorithms for current hardware architectures, although these algorithms are considered hard to parallelize [54].

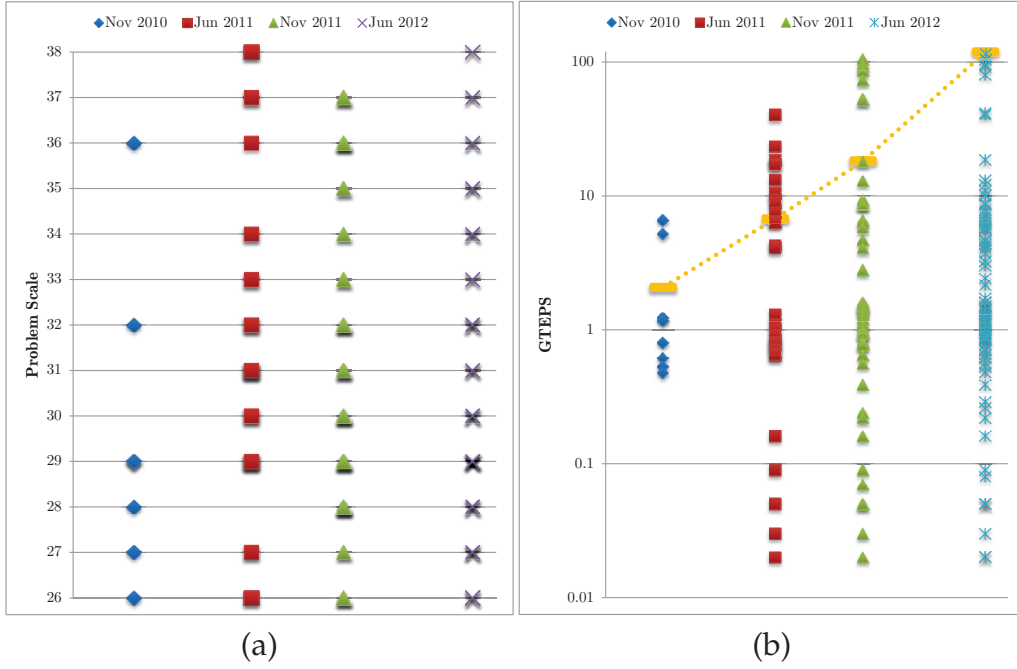
Most graph algorithms follow a type of bulk synchronous parallel programming [31, 235] pattern: computation phases are succeeded by synchronization phases.

A further characteristic of these algorithms is the very low amount of computation per loaded byte, and the high number of communication messages. Although the speed of network interconnections has increased, latency in the networks is a crucial factor for the scalability of graph algorithms on current parallel architectures.

Common issues a parallel algorithm designer is faced with are irregular memory access patterns (i.e., unavailability of a closed-form expression for the subscripts of the accessed variable at compile time), unbalanced load of data, and the frequent use of synchronization primitives. A low spatial locality — i.e., a memory location close to referenced data at a particular time is unlikely to be referenced in the near future — and rare temporal locality — i.e., a particular memory location is unlikely to be referenced again in the near future — are the logical consequences. Thus, many cache misses are typical inconveniences during the execution of graph algorithms. Furthermore, it is hard to predict future memory accesses, as the control flow of a graph algorithm is determined by the input data.

A small number of high performance libraries and software toolkits, such as, SNAP [18], Combinatorial BLAS [39], ColPack [58], and Parallel BGL [106], are available to solve occurring graph problems in data intensive applications.

In order to benchmark graph algorithms running on supercomputers for data intensive applications, the Graph 500 list has been published at Supercomputing 2010 [225]. The benchmarks consist currently of two



**Figure 1.1:** Visualization of benchmark data of three Graph 500 lists.

compute kernels, construction of a graph via a Kronecker generator and its graph traversal via e.g., breadth-first search algorithms. The input sizes of the problems are classified into 6 levels: from 17 GB up to 1.1 PB. In order to compare the performance of the algorithms across a variety of architectures, programming models, and frameworks, the performance metric TEPS has been introduced which counts the number of Traversed Edges Per Second; the higher the TEPS on a supercomputer, the higher ranked is the machine.

In Fig. 1.1, available data are visualized for the dates November 2010, June 2011, November 2011, and June 2012. In Fig. 1.1(a), the base-2 logarithm numbers of the vertices in the graph (starting with  $2^{25}$  up to  $2^{38}$ ) are illustrated for these four dates. Since the interest in solving graph algorithms on supercomputer grows — indicated by the fact that the number of entries in the lists increased from 9 (Nov 2010) to 80 (Jun 2012), a broad spectrum of problem classes could be solved on current supercomputers. The largest graph benchmarked so far requires a storage size of more than 100 TB. In Fig. 1.1(b), the number of traversed edges in billions (GTEPS) is presented for the existing data. The yellow bars represent the average number of GTEPS which exponentially grows over the period of time. Additionally, the range of GTEPS obtained has been enlarged as more

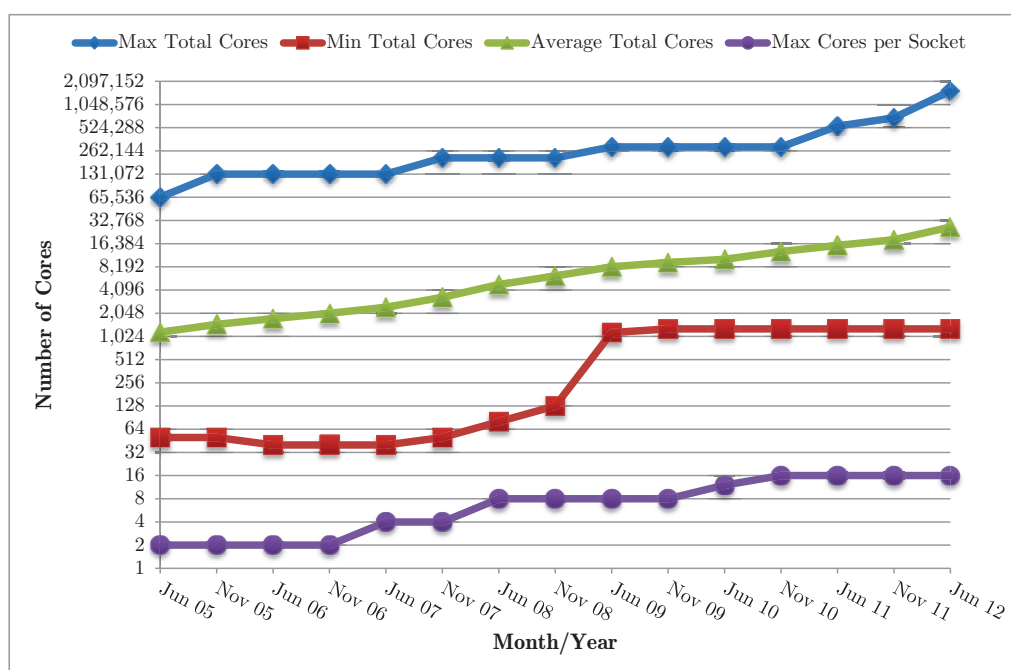
and more researchers put some efforts to benchmark the algorithm on a supercomputer or rather small multicore cluster machines. It can be expected that the benchmark results will attract more attention in the near future and will be populated with more data in the subsequent releases. There is also a strong trend towards the next unit TTEPS, traversed edges per second in trillions.

The key role of graph algorithms in data intensive applications [154] can also be seen by the increasing number of conferences and publications: since 2004, on a regular two-year basis, a SIAM workshop devoted to combinatorial graph problems in computational science and engineering has been organized where international researchers present sequential and parallel graph algorithms and their applications [179]. The growing interest in combinatorics combined with computational science has also been credited with the first Dagstuhl seminar and with a book based on the meeting [174].

The second essential part to analyze large graphs is high performance computing. High performance computing characterizes a multidisciplinary research field covering parallel algorithm engineering, parallel architectures, parallel programming, and the solving of emerging applications, in particular, in computational science.

The challenge for a parallel algorithm engineer is the design and development of scalable algorithms on massively parallel architectures. The notion “scalable” refers to the compute performance of the algorithm: assuming a parallel algorithm runs in  $T_p$  seconds on a parallel architecture with  $P$  cores and converges in  $T_s$  seconds on a single core. The algorithm scales well on the given architecture if the time of the algorithm drops with the number of compute cores. Ideally, the time of the algorithm reduces linearly by using  $p \leq P$  cores. Testing the linear speedup of the algorithm with  $T_s/T_p$  refers to “strong” scalability and is the most relevant measure in practice. An algorithm that scales linearly with the number of cores in theory is categorized as an embarrassingly parallel algorithm. Graph algorithms do not fall into this category of algorithms and as a consequence most graph algorithms are designed to run on uniprocessors.

Since 1965, transistors on a chip double every 18 – 24 months (which is known as Moore’s Law), but since 2004, clock frequency and compute performance of uniprocessors have stagnated due to the need to reduce voltage. Consequently, the computer industry has moved from the development of uniprocessors to multicore processors which typically feature



**Figure 1.2:** Visualization of recent trends in the TOP500 Supercomputing list.

clock rates between 2 and 3 GHz. In current supercomputers, multiple multicore processors are connected in order to construct a scalable high performance system.

With supercomputers getting increasingly faster, software ported to these cluster machines is useful for many real life applications which could not efficiently be tackled before. Since June 1993, the TOP500 Supercomputing list [226] is released twice per year, which corroborates trends of advancing hardware technologies, and ranks the fastest supercomputers worldwide. The term “fastest” does not refer to the theoretical maximum performance of the system, but to the performance measured by the LINPACK benchmark. In this benchmark, a dense system of linear equations is solved with direct methods using Gaussian elimination for the respective hardware architecture. It reflects a reliable measure of the system performance for a commonplace problem and, hence, provides a practical and realistic estimation of the machine’s actual performance. In Fig. 1.2, the maximum, minimum, and average number of compute cores of the supercomputer, and the maximum number of compute cores per socket are illustrated based on the 500 fastest supercomputers at the evaluation date starting with June 2005. Over the seven year period, the average number of compute cores doubled almost every year.

Remarkably, the maximum number of cores of the currently\* fastest 16.32 petaflop supercomputer, the SEQUOIA (IBM, USA), is more than one order of magnitude larger than the maximum number of cores of BLUE-GENE/L (IBM, USA) six years before. The minimum number of cores of the TOP500 supercomputers was relatively constant up to November 2008. At that point, a significant jump in the number of minimum cores occurred. This phenomenon can be explained by the increasing availability of multicore processors. In 2005 and 2006, a socket of a supercomputer hosted at maximum a dual-core processor; thenceforth, the number of cores per socket has roughly doubled every year.

Today, a socket usually contains multiple cores and is integrated into compute nodes which feature several sockets. Usually, each socket is equipped with its own shared memory including their complex memory hierarchies. Each node also features shared memory across multiple sockets. All nodes are interconnected via fast interconnection networks with other nodes in a specific topology, a popular choice being a 3-D torus.

Parallel programming is the key to address these massively parallel distributed and shared memory architectures. On the one hand, computational workload should be distributed among compute nodes, and on the other hand, computation on a compute node should be ultimately accelerated using its compute cores.

Communication between compute nodes is mostly established by a message passing system, for which the Message Passing Interface (MPI) provides a de facto standard [86]. Normally, each node is assigned to one MPI process, performs computation independently of other processes, and whenever a data exchange is necessary, MPI offers several communication constructs such as point-to-point communication or collective communication functions, to make data available across nodes.

From a programmer's point of view, favorite programming languages like C, C++, Fortran, or Java can still be used to write source code, but this sequential source code is interspersed with MPI function calls whenever communication is needed. As MPI can establish a mapping of processes to nodes, acceleration of the computation on a node is achieved by shared memory programming. A widely used multithreading API for shared memory parallelization is OpenMP [180]. It can be easily integrated into the existing source code by adding directives to sections which are

---

\*TOP500 list from June 2012



intended to run in parallel. Thus, master and slaves threads are created at the beginning of the parallel region, and incorporate the fork-join principle. Both APIs are designed for both task and data parallelism. Today, the hybrid programming model, MPI combined with OpenMP, is one of the most widely applied models to achieve maximum performance by a supercomputer. Beyond MPI and OpenMP, a promising parallel programming model is the partitioned global address space supported by languages like Unified Parallel C, Titanium, Chapel, or X10 [50]. Recent supercomputers contained hardware accelerators like graphics processing units (GPUs) [105] on a node which may enhance performance of software if data parallelism constitutes a dominant part of the computation. Although graph algorithms are commonly not compute intensive, it is an open question if accelerators can speed up graph algorithms [111, 168].

The focus of this thesis is on the following graph problems: graph matching, graph partitioning, and the dense subgraph problem.

Consider, as an example for bipartite graph matching, the stable marriage problem: assume two disjoint data sets of  $n$  men and  $n$  women are given and their relations are measurable by a number. Then, a so-called bipartite graph can be constructed where each man is represented by a vertex in the left part of the graph, and each woman by a vertex in the right part of the graph. An edge connects two vertices if the corresponding man and woman like each other, and the weight of the edge quantifies the depth of their mutual interest in each other. The question arises whether it is possible to find pairs of men and women subject to nobody remains unassigned and everybody is satisfied with the designated mate. Solving this issue with a brute-force algorithm which samples all possible assignments will require testing  $n!$  assignments to obtain the optimal solution. However, as the size of the matching problem  $n$  grows to millions of vertices, the complexity of this approach is beyond any practical scope. But clever algorithms have been developed which find the optimal solution of the bipartite matching problem in polynomial time [104, 142, 173].

The second problem is graph partitioning where the task is splitting a graph into several disjoint parts subject to the number of vertices in each part being almost equal and the number of heavy-weighted edges between parts being minimized. Today, there are many fast and efficient heuristics to solve the problem approximately but, in general, none of them guarantees finding an optimal solution [125, 229].

The third problem discussed in the thesis is the finding of subgraphs in a large graph where a subgraph with  $k$  vertices will be discovered in

a graph with  $n$  vertices ( $k \ll n$ ) subject to the weight of the subgraph being the largest among all subgraphs of size  $k$  in the entire graph. Efficient algorithms are also only known to compute a solution of suboptimal quality [81].

The basic motivation behind the focus on these graph algorithms is that the algorithms play a significant role speeding up the simulation and optimization of data intensive applications in numerical linear algebra [72, 129]. For instance, in life science applications, the simulation of arterial flow is of considerable interest since a disease like aortic aneurysm cannot be treated efficiently without using simulation software [216]. As the mathematical modeling in scientific simulations such as fluid flow can be described by a system of partial differential equations, one major time consuming part in numerical software toolkits lies in the computation of solutions of large sparse linear equation systems. Consequently, accelerating solving sparse linear systems will enhance the speed of the entire simulation process. A natural way to speed up the computation of both, direct and iterative, linear solvers is to design the solvers for multicore clusters [6, 108, 208]. A recent advance in the research field is to combine direct with iterative linear solvers to obtain a so-called hybrid linear solver. The hybrid linear solver PSPIKE is one instance of these new classes of solvers [162] which need solutions of efficient parallel graph algorithms for the bipartite graph matching, the graph partitioning, and the dense subgraph problem as a preprocessing step.

The thesis is organized into five parts. The first part introduces existing concepts and methods as a foundation to the subsequent parts which will describe the key thesis contributions in detail. It surveys graph problems and existing algorithms in computational science and, in particular, the principle of auctions to solve the bipartite graph matching problem. In part two, the parallel auction algorithm implementation PAUL is presented and implementation aspects are discussed. The third part motivates the need to solve a dense subgraph problem in the hybrid linear solver PSPIKE and proposes different heuristics to find weighted subgraphs. Furthermore, implementation aspects of PSPIKE are described with an emphasis on the integration of graph algorithms into the solver. In the fourth part, PSPIKE and PAUL are applied to data intensive applications like arterial flow simulation, optimal control of partial differential equations, and graph similarity in protein-protein interaction networks. Finally, part five contains the conclusion and the outlook.

## 1.1 Contributions

The main contributions of this thesis are:

- Designing of an auction-based weighted matching algorithm for parallel distributed-memory architectures using the MPI–OpenMP programming model
- Introducing different  $\varepsilon$ -scaling strategies in the parallel auction algorithm PAUL
- Finding weighted matchings for bipartite graphs which can be either dense or sparse, and either balanced or unbalanced
- Identifying a new weighted dense subgraph problem in PSPIKE and proposing different efficient heuristics to solve the  $\mathcal{NP}$ -hard problem; the most effective heuristic is based on a  $(1 + 1)$  evolutionary algorithm
- Integrating several strong reordering schemes into the preprocessing phase of the hybrid linear solver PSPIKE which are based on solutions of the graph matching, graph partitioning, and weighted dense subgraph problem
- Solving the entire inner dense linear systems in PSPIKE with a preconditioned iterative linear solver; the accuracy of the solution can be adapted to the given application
- Including two features into PSPIKE that allow input of a known good preconditioner and a starting solution
- Validating PAUL and PSPIKE on several data intensive applications like arterial flow dynamics, PDE-constrained optimization, and graph similarity
- Implementing PAUL and PSPIKE as standalone parallel software libraries which are available at <http://www.pspike-project.org>.



## **Part I**

# **Graph Theory in Computational Science**



## Chapter 2

---

# Graph Problems and Algorithms

---

### 2.1 Preliminaries

In this section fundamental concepts of graph theory are introduced [129].

A general undirected *weighted graph*  $\mathcal{G}$  is a quadruple  $\mathcal{G} = (\mathcal{V}, \mathcal{E}, w, c)$ , with  $\mathcal{V} = \{1, \dots, n\}_{n \in \mathbb{N}}$ ,  $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ ,  $w : \mathcal{E} \rightarrow \mathbb{R}$ , and  $c : \mathcal{V} \rightarrow \mathbb{R}$ . The weight  $w_{ij} := w(i, j)$  indicates the weight of edge  $e = (i, j)$  with  $i, j \in \mathcal{V}$ ,  $c_i := c(i)$  denotes the weight of vertex  $i$ . Endpoints  $i$  and  $j$  of edge  $e$  are called *adjacent* to each other and vertices  $i, j$  are referred to as *incident* to the edge. Edges are called adjacent to each other if they share a common vertex. The degree of vertex  $i$ ,  $\deg(i)$ , is the number of edges incident to  $i$ .

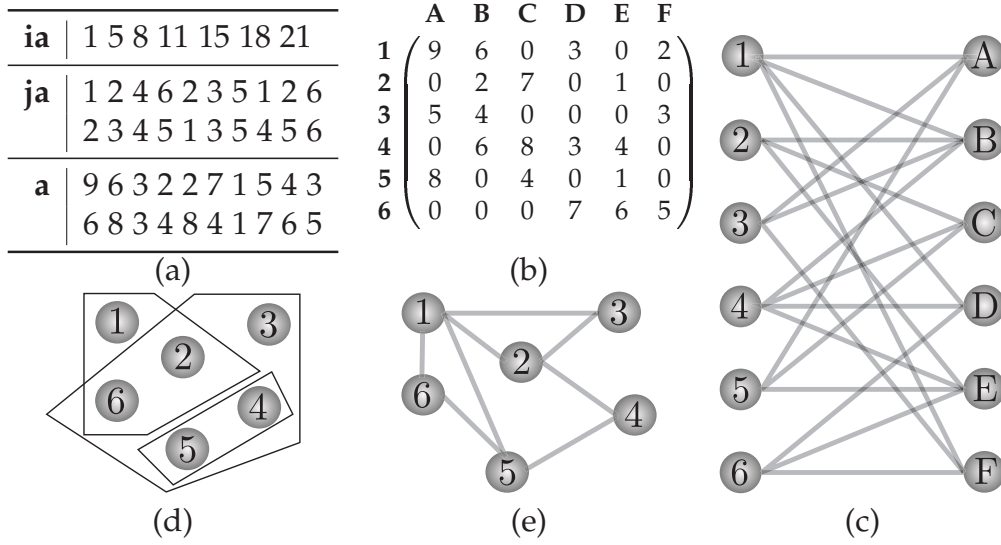
A *path* in  $\mathcal{G}$  is a sequence of vertices  $[v_1, v_2, \dots, v_k]$  of  $\mathcal{V}$  with  $k \geq 2$  and a corresponding sequence of  $k - 1$  edges of the form  $(v_i, v_{i+1})$ .

A graph  $\mathcal{H} = (\mathcal{U}, \mathcal{F}, w, c)$  is a *subgraph* of  $\mathcal{G}$  iff  $\mathcal{U} \subseteq \mathcal{V}$  and  $\mathcal{F} \subseteq \mathcal{E}$ . Two graphs  $\mathcal{G}_1 = (\mathcal{V}_1, \mathcal{E}_1)$  and  $\mathcal{G}_2 = (\mathcal{V}_2, \mathcal{E}_2)$  are *isomorphic* if there is a bijective function  $f : \mathcal{V}_1 \rightarrow \mathcal{V}_2$  such that  $(i, j) \in \mathcal{E}_1 \Leftrightarrow (f(i), f(j)) \in \mathcal{E}_2$  for all  $i, j \in \mathcal{V}_1$ .

A *matching*  $\mathcal{M}$  in graph  $\mathcal{G}$  is a subset of  $\mathcal{E}$ ,  $\mathcal{M} \subseteq \mathcal{E}$ , where edges in  $\mathcal{M}$  are pairwise nonadjacent. Edges in  $\mathcal{M}$  and their adjacent vertices are called *matched* edges and *matched* vertices, respectively. Edges, which are not in  $\mathcal{M}$ , and vertices which are not endpoints of a matched edge are called *free*.

A *partition* of  $G$  splits the graph into nonempty subgraphs  $V_l$  subject to  $\bigcup_{l=1}^K V_l = \mathcal{V}$  and  $V_{l_1} \cap V_{l_2} = \emptyset$  for all  $1 \leq l_1 < l_2 \leq K$ .

A *bipartite graph*  $\mathcal{G}_b = (\mathcal{V}_1, \mathcal{V}_2, \mathcal{E}, w)$  is a bipartition ( $K = 2$ ) of  $\mathcal{G}$  with vertex sets  $\mathcal{V}_1, \mathcal{V}_2$ ,  $|\mathcal{V}_1| = n_1$ ,  $|\mathcal{V}_2| = n_2$ , and edge set  $\mathcal{E} \subseteq \mathcal{V}_1 \times \mathcal{V}_2$ . A



**Figure 2.1:** Different graph and matrix representations. Top: CSR matrix (a), corresponding adjacency matrix (b), and unweighted bipartite graph (c). Bottom: a hypergraph (d) and a general graph (e).

bipartite graph is called *balanced* if  $n_1 = n_2$  and otherwise referred to as *unbalanced*. If not otherwise indicated,  $n_1 \leq n_2$ ,  $n = n_2$ , and  $|\mathcal{E}| = m$ .

A hypergraph  $\mathcal{G}_h = (\mathcal{V}, \mathcal{E}_h, w, c)$  is a generalization of  $\mathcal{G}$  where every hyperedge  $e_h \subseteq \mathcal{V}$ ,  $\bigcup e_h = \mathcal{E}_h$ ,  $w : \mathcal{E}_h \rightarrow \mathbb{R}$  with  $w_i$  the weight of hyperedge  $i$ , and  $c : \mathcal{V} \rightarrow \mathbb{R}$  with  $c_j$  the weight of vertex  $j$ .

Every graph  $\mathcal{G}$  can be represented by its weighted *adjacency matrix*  $\mathcal{A}^{n_1 \times n_2}$  where nonzero entry  $a_{ij} = w_{ij}$ . Matrix  $\mathcal{A}$  is *sparse* if most of the entries are zero. Otherwise  $\mathcal{A}$  is called *dense*.

A sparse matrix  $\mathcal{A}$  can be stored quite efficiently in common sparse storage formats; compressed sparse row (CSR), compressed sparse column (CSC), and coordinate list (COO) are widespread approaches to substantially reduce the memory requirements with respect to dense storage formats. A natural way of keeping entries is COO, as the row and column index of each entry are explicitly stored together with the numerical value. Thus, the storage consumption of COO is  $3m$ . The memory usage can be further reduced by CSR where row indices are substituted by an array “ia” of row pointers, requesting memory of  $n + 1$ . Thus, the entire matrix requires memory of  $n + 1 + 2m$ . In contrast to CSR, CSC storage format keeps column pointers, row indices, and numerical values. The transpose of a CSR matrix is the CSC format of the sparse matrix  $\mathcal{A}$ . In Fig. 2.1, different representations of the same matrix/graph are given in



(a)–(c), while a hypergraph in (d) and a general graph in (e) are visualized. In (a), the array with the row pointers is given by “ia,” the array with the column indices is presented by “ja,” and the numerical values are stored in the array “a.”

The adjacency matrix  $\mathcal{A}$  of a balanced bipartite graph is a *square* matrix, and  $\mathcal{A}$  is *rectangular* if the graph is unbalanced. It requires memory of  $\mathcal{O}(n^2)$ . Matrix  $\mathcal{A}$  is *symmetric* if  $\mathcal{A} = \mathcal{A}^\top$ , and otherwise *unsymmetric*. Every unsymmetric matrix  $\mathcal{A}$  can be transformed into a symmetric matrix by  $\tilde{\mathcal{A}} = \mathcal{A} + \mathcal{A}^\top$ .

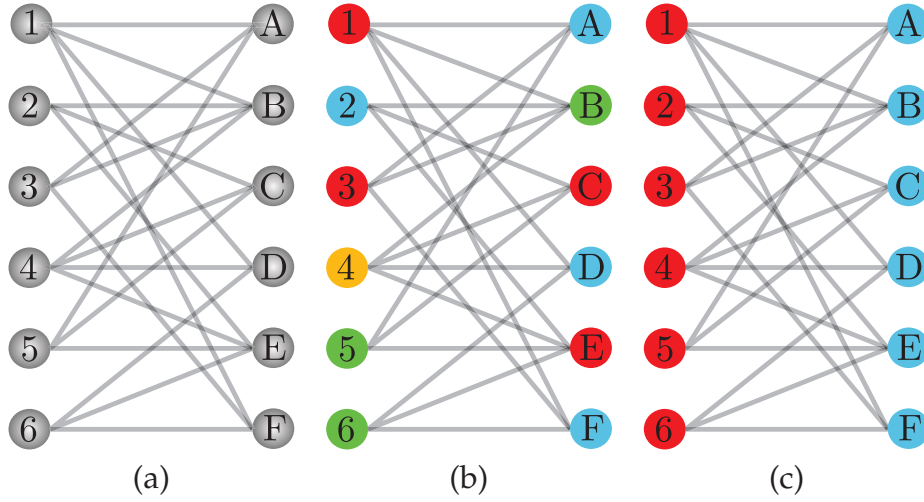
Suppose  $\tilde{\mathcal{A}}$  is a symmetric matrix with a zero-free diagonal. Let  $b_j = j - \min\{i : \tilde{a}_{ij} \neq 0\}$ ; i.e.,  $b_j$  is the distance between the first nonzero entry in column  $j$  and the diagonal. Then, the *profile* and the *bandwidth* of  $\tilde{\mathcal{A}}$  are defined as  $\sum_j b_j$  and  $\max_j b_j$ , respectively. The  $n \times n$  Laplacian  $L = (l_{ij})$  of  $\tilde{\mathcal{A}}$  is defined by

$$l_{ij} := \begin{cases} \deg(v_i) & \text{if } i = j, \\ -1 & \text{if } i \neq j \text{ and } \tilde{a}_{ij} \neq 0, \\ 0 & \text{otherwise.} \end{cases}$$

### 2.1.1 Combinatorial Graph Problems

In computational science and engineering, a rich fund of sparse and dense graph problems is available ranging from well-known problems such as graph partitioning, graph matching, traversals, and graph coloring to lesser addressed problems such as subgraph problems and routing [174].

In automatic differentiation [107], the computation of a sparse Jacobian or Hessian using graph coloring techniques is a well established approach. The task in vertex coloring is to assign a minimum number of different labels (colors) to vertices subject to certain constraints. An important problem is the distance- $k$  coloring problem. The goal is to find a mapping  $g : \mathcal{V} \rightarrow \{1, 2, \dots, o\}$  with the objective to minimize  $o$  — the number of colors — subject to  $g(i) \neq g(j)$  whenever vertices  $i, j$  are distance- $k$  neighbors. Two vertices  $i$  and  $j$  are distance- $k$  neighbors if there is path of length  $k$  connecting the vertices. For instance, the distance-1 coloring problem assigns different colors to adjacent vertices with the objective to minimize the number of colors. The minimum number of colors is known as the chromatic number and is denoted by  $\chi(\mathcal{G})$ . In general, the distance- $k$  graph coloring problem is  $\mathcal{NP}$ -hard, but researchers actively design parallel graph coloring heuristics on parallel architectures [92]. A simple greedy heuristic to solve the distance-1 coloring problem can



**Figure 2.2:** Example for distance-1 vertex coloring using the greedy heuristic with different orderings. (a) bipartite graph  $\mathcal{G}_b$ , (b) coloring with  $\chi(\mathcal{G}_b) = 4$ , (c) coloring with  $\chi(\mathcal{G}_b) = 2$ .

be stated as follows. At the beginning each vertex is uncolored. Then, visit the vertices in some order and assign a “feasible” color to the vertex, where a color is feasible if it is not used by any adjacent vertex. If there is no feasible color available, a new color is introduced, and the vertex is labeled with this new color. The time complexity of this algorithm is  $\mathcal{O}(n + m)$  since every vertex is visited at least once and, additionally, the feasibility of the color must be checked. The minimum number of colors found by the greedy heuristic is bounded by  $\chi(\mathcal{G}) \leq \Delta(\mathcal{G}) + 1$ , where  $\Delta(\mathcal{G})$  denotes the maximum degree of a vertex in  $\mathcal{G}$ .

In Fig. 2.2, the distance-1 vertex coloring problem is solved by using the greedy heuristic. In Fig. 2.2(a) the input graph is illustrated, whereas in Figs. 2.2(b) and (c) the result of the greedy heuristic is highlighted when applied to the input graph. In Fig. 2.2(b), the vertices are traversed in the ordering (3, C, 2, B, 1, A, 4, D, 5, E, 6, F), whereas in Fig. 2.2(c) the ordering (1, A, 2, B, 3, C, 4, D, 5, E, 6, F) finds the minimal number of colors. It can be concluded that the ordering of the vertices highly influences  $\chi(\mathcal{G}_b)$ .

An improvement in the quality of the heuristic is to compute a maximal independent set MIS in the graph in polynomial time as colored vertices build an independent set [153]. Then, these vertices are colored with the same color and the set is removed from the graph along with the adjacent edges. This procedure is repeated until the graph is empty.

In social network analysis, graph theoretical metrics such as network density, network centrality, and cluster analysis are of growing interest. For instance, betweenness centrality helps to identify the most influential persons in terrorists networks [140], and cluster analysis detects communities in social networks which share some common properties [46, 100].

The computation of betweenness centrality is dependent on the solution of the shortest path problem as vertices lying on many shortest paths are considered to be more important and have a higher betweenness than other vertices. Given a source vertex  $v_s$  and a destination vertex  $v_d$  in  $\mathcal{G}$ , a shortest path is a path  $[v_s, v_{s+1}, \dots, v_{d-1}, v_d]$  with minimum distance, where distance is defined as the sum of the absolute weights of the incident edges on the path. One famous algorithm to solve the single-source shortest path problem is Dijkstra’s algorithm which can be implemented using Fibonacci heaps in  $\mathcal{O}(m + n \log n)$  [67, 87]. However, Dijkstra’s algorithm is inherently sequential as vertices are visited in a fixed priority order. The  $\Delta$ -stepping algorithm provides a parallel implementation of Dijkstra’s idea which could be successfully ported to massively parallel architectures [156, 169]. For arbitrary graphs, the sequential  $\Delta$ -stepping has a complexity of  $\mathcal{O}(n + m + \Delta(\mathcal{G})F)$ , where  $\Delta(\mathcal{G})$  is the maximum vertex degree in the graph and  $F$  denotes the maximum weight of a shortest path from  $s$  to any vertex reachable from  $s$ .

In computational biology and chemistry, graph theory enables the modeling of emerging complex networks, and provides fast graph algorithms and heuristics in order to better understand the topology and function of the networks. Thus, molecules, proteins, and sequences are typically modeled as trees or graphs [122] as, e.g., the secondary RNA structure of species [24, 90] and the amino acids of proteins [11]. An important aspect for researchers is that biological data are often stored in numerous, free accessible databases like EMBL [143] and DIP [237]. Additionally, algorithm engineers practice their algorithms on artificial graphs like Erdős-Rényi graphs [80] which randomly generate edges between vertices resulting in random graphs with similar properties (e.g., having the power-law distribution of the biological networks with their noisy data).

Crucial graph problems that must be efficiently treated are based on statistical measurements like centrality (e.g., degree centrality), finding of “Motifs” — which are subgraphs that occur often in the network, finding several paths between vertices to check the robustness of the network, finding clusters (e.g., overlapping or highly connected clusters) to iden-

tify functional modules and dependencies in the network, performing alignments across multiple sequences, and visualization of the dynamical behavior of a large-scale network [4].

Further important graph problems which frequently occur in diverse application domains are graph matching, graph partitioning, and the dense subgraph problem, which is the focus of this thesis, which are discussed in more detail in the following.

## 2.2 Graph Matching Problem

The graph matching problem is one of the oldest combinatorial optimization problems and has been studied for almost a century [110, 137]. Identifying weighted matchings is a kernel computation in a wide range of applications in bioinformatics [141, 157, 218, 224, 236, 240], computer vision [23, 55, 152, 213], sparse linear algebra [73, 109], other combinatorial problems [16, 45, 167], and in other areas of computational science [52, 79].

### 2.2.1 Weighted Graph Matching

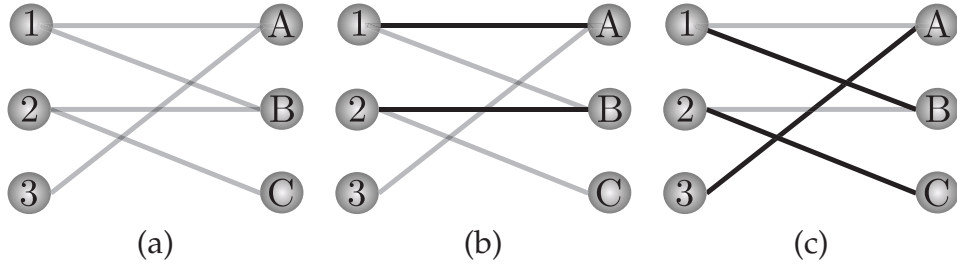
Formally, a subset  $\mathcal{M} \subseteq \mathcal{E}$  in a bipartite graph  $\mathcal{G}_b$  is called matching iff  $|\mathcal{M}| = 1$  or

$$(v_1, w_1) \in \mathcal{M} \wedge (v_2, w_2) \in \mathcal{M} \Rightarrow (v_1 \neq v_2) \wedge (w_1 \neq w_2), \quad (2.1)$$

where  $v_1, v_2 \in \mathcal{V}_1$  and  $w_1, w_2 \in \mathcal{V}_2$  and  $(v_1, w_1) \neq (v_2, w_2)$ . The total weight of the matching is computed either by  $W_1 = \sum_{(i,j) \in \mathcal{M}} |w_{ij}|$  or by  $W_2 = \prod_{(i,j) \in \mathcal{M}} |w_{ij}|$ .

A large number of matching algorithms are designed to achieve a matching which maximizes the cardinality of  $\mathcal{M}$  and often the weight of the matching, simultaneously.

In a *maximal* matching, no edge can be added to  $\mathcal{M}$  without violating the matching property given by (2.1). A *maximum* (cardinality) matching is a matching which contains the largest possible number of edges. The maximum cardinality matching problem asks for a matching  $\mathcal{M}$  that contains the maximum number of edges. If  $\mathcal{M} = |\mathcal{V}_1| = |\mathcal{V}_2|$ , such a matching  $\mathcal{M}$  is called perfect, where  $|\cdot|$  denotes the cardinality of a set [151]. Clearly not all bipartite graphs have a perfect matching. Note that every maximum matching is a maximal matching, but the converse is not



**Figure 2.3:** Difference between maximal (b) and maximum — here also perfect — matching (c) of a balanced bipartite graph (a).

true, in general. According to the theorem of Berge [151], a matching is a maximum matching if it contains no *augmenting paths*. A path  $P_A$  in  $\mathcal{G}_b$  is called  $\mathcal{M}$ -*augmenting* if  $P_A$  has odd length, if the edges of  $P_A$  are alternatively in and not in  $\mathcal{M}$ , and if the end vertices are not covered by  $\mathcal{M}$ . Then, the symmetric difference  $\mathcal{M} \Delta P_A := (\mathcal{M} \cup P_A) \setminus (\mathcal{M} \cap P_A)$  is a matching and  $|\mathcal{M} \Delta P_A| = |\mathcal{M}| + 1$ . There are many different algorithms for computing maximum or maximal matchings in a bipartite graph, e.g., [72, 116, 147, 184]. In Fig. 2.3, the difference between a maximal (b) and a maximum matching (c) is shown. In Fig. 2.3(b), the edge (3, A) cannot be added to the matching as vertex A is already assigned to vertex 1. In Fig. 2.3(c), a perfect matching is shown as all vertices are matched in the matching.

In this thesis, algorithms will be studied that also take the weight of the edges into consideration and which either find a maximum matching with a high weight [28, 88, 142, 173], in short, *maximum weighted matching*, or a maximal matching with a high weight [193], in short, *maximal weighted matching*. Most algorithms for the maximum weighted matching problem search first for a maximum cardinality matching, and additionally maximize the edge weights in the matching, whereas most algorithms for the maximal weighted matching problem prefer to maximize the weight of the matching, and then the cardinality of the matching.

### 2.2.2 Landscape of Graph Matching Algorithms

A variety of *approximation* algorithms and *exact*, polynomial-time algorithms have been designed to find a matching.

Approximation algorithms return a maximal weighted matching. For instance, a greedy algorithm (see Algorithm 2.1) can be described as follows. Sort the weights of the edges in a list in decreasing order. Then,

**Algorithm 2.1:** *Greedy Heuristic for Weighted Graph Matching*

**Input:** Bipartite graph  $\mathcal{G}_b = (\mathcal{V}_1, \mathcal{V}_2, \mathcal{E}, w)$   
**Output:** Matching  $\mathcal{M}$

```

1:  $\mathcal{M} \leftarrow \emptyset$ 
2: Sort  $|w_{ij}|$  by decreasing weight and store in list  $L$ 
3: while ( $|L| > 0$  and  $|\mathcal{M}| < n_1$ ) do
4:   Take heaviest edge  $(i, j)$  in  $L$ 
5:   if  $i$  and  $j$  are not matched vertices then
6:      $\mathcal{M} \leftarrow \mathcal{M} \cup (i, j)$ 
7:   end if
8:   Remove edge  $(i, j)$  from  $L$ 
9: end while

```

select the heaviest edge  $e$ , and check if the endpoints of the edge are not matched. If both are free, add the edge to the matching set, and delete the edge from the list. The selection and deletion process is repeated until the list is empty or a maximum matching has been attained. The worst case running time of the greedy approach is  $\mathcal{O}(m \log m)$ . This simple algorithm has an approximation factor of  $1/2$  [193]. Sophisticated approaches such as a  $2/3$ - or  $3/4$ -approximation have been published by several authors [70, 71, 188, 230]. Attempts to parallelize  $1/2$ -approximation algorithms have been described in [49, 117, 163, 184].

In contrast to approximation algorithms, exact algorithms guarantee to find a maximum matching. Most methods compute a maximum matching using the concept of augmenting paths and are inspired by maximum flow algorithms due to the fact that a bipartite graph can be represented as a flow network by introducing source and sink vertices, and by transforming undirected edges to directed edges pointing from  $\mathcal{V}_1$  to  $\mathcal{V}_2$  [41].

Many algorithms compute a maximum cardinality matching irrespective of the weight by finding shortest augmenting paths via a depth-first search or a breadth-first search [17]. One of the fastest known algorithms is the push-relabel algorithm with a worst-case running time of  $\mathcal{O}(\sqrt{nm})$  [72, 104]. Recently, a parallel implementation of the push-relabel algorithm has been derived for distributed memory architectures [146].

For finding a maximum weighted matching, the Hungarian method is a popular algorithm with a running time of  $\mathcal{O}(n(m + n \log n))$  [87, 142, 173]. Fast, but inherently sequential, shortest augmenting path imple-

mentations are provided, for instance, by the routine MC64 in the HSL library, which has a running time of  $\mathcal{O}(n(m+n)\log n)$  [73], or in [120] to find maximum weighted matchings.

Solving the perfect weighted matching problem is also known as the *linear sum assignment problem* in combinatorial optimization [41]. It can be formulated as

$$\begin{aligned}
 \max \quad & \sum_{(i,j) \in \mathcal{E}} w_{ij} x_{ij} \\
 \text{s.t.} \quad & \sum_{i \in \mathcal{V}_1} x_{ij} = 1 \quad \text{for } j \in \mathcal{V}_2, \\
 & \sum_{j \in \mathcal{V}_2} x_{ij} = 1 \quad \text{for } i \in \mathcal{V}_1, \\
 & x_{ij} \geq 0.
 \end{aligned}$$

The dual problem is equivalent to

$$\begin{aligned}
 \min \quad & \sum_{i=1}^{n_1} r_i + \sum_{j=1}^{n_2} p_j \\
 \text{s.t.} \quad & r_i + p_j \geq w_{ij} \quad \text{for } (i,j) \in \mathcal{E},
 \end{aligned}$$

where  $r$  and  $p$  are the dual variables.

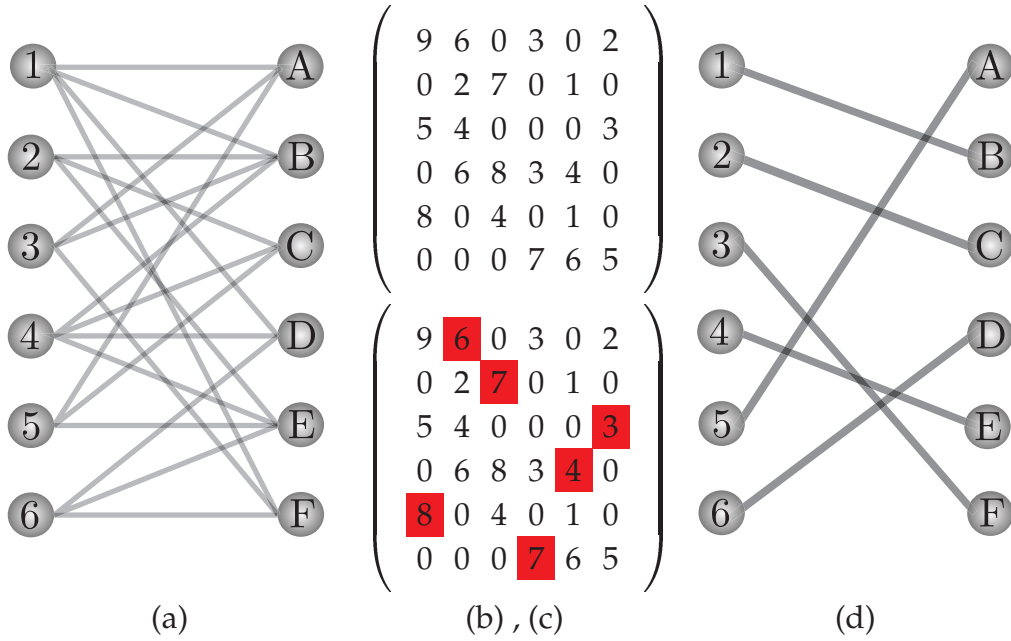
Linear programming techniques such as simplex algorithms and interior point methods can be applied to the primal or dual program to attain a feasible assignment [41, 103].

Due to the modeling of the problem as a linear program, most maximum weighted matching algorithms are also called *primal dual algorithms* as they typically compute a feasible dual and infeasible primal solution and iteratively update them until the algorithm is converged to a feasible primal solution.

In Fig. 2.4(a), a perfect weighted matching is computed on a bipartite graph of size 6. The matching is highlighted in red in the weighted adjacency matrix (see Figs. 2.4(b), (c)), and the matched edges are shown in Fig. 2.4(d).

Real-world economic auctions serve as a metaphor for a major class of maximum weighted matching algorithms, called *auction algorithms*, which are analyzed for parallelization and, therefore, described in detail in Chapter 3.





**Figure 2.4:** Matching illustration: input graph (a) and adjacency matrix (b), matching output (c), (d) with  $W_1 = 35$ .

## 2.3 Graph Partitioning and Ordering

Especially in parallel computing, graph partitioning and ordering play a central role in data intensive scientific applications such as sparse matrix–vector multiplication [229], finite element methods [132], data clustering [119], VLSI design [127], and fill-in reduction in direct linear solvers in numerical linear algebra [62, 208]. The goals in such applications are to distribute the same amount of data among processes and to reduce the need for data exchange by minimizing the number of adjacent vertices to different processes. Both objectives can be achieved by graph partitioning algorithms with the objective to minimize the amount of communication subject to balancing computational work among processes.

### 2.3.1 Graph and Hypergraph Partitioning Models

Given a graph  $\mathcal{G}$ , the constraint to balance data among processes can be expressed as

$$\sum_{i \in \mathcal{V}_l} c_i \leq \frac{(1 + \varepsilon_P)}{K} \sum_{j \in \mathcal{V}} c_j \quad \text{for } l = 1, \dots, K, \quad (2.2)$$



where  $\varepsilon_P \geq 0$  is the imbalance parameter and  $c_i$  is the cost of vertex  $i$ .  $c_i$  can be interpreted as, e.g., the number of adjacent vertices.

A partitioner aims at minimizing the *edge cut*  $\chi_1(\Pi)$  between partitions  $\mathcal{V}_{l_1}, \mathcal{V}_{l_2} \in \Pi$ :

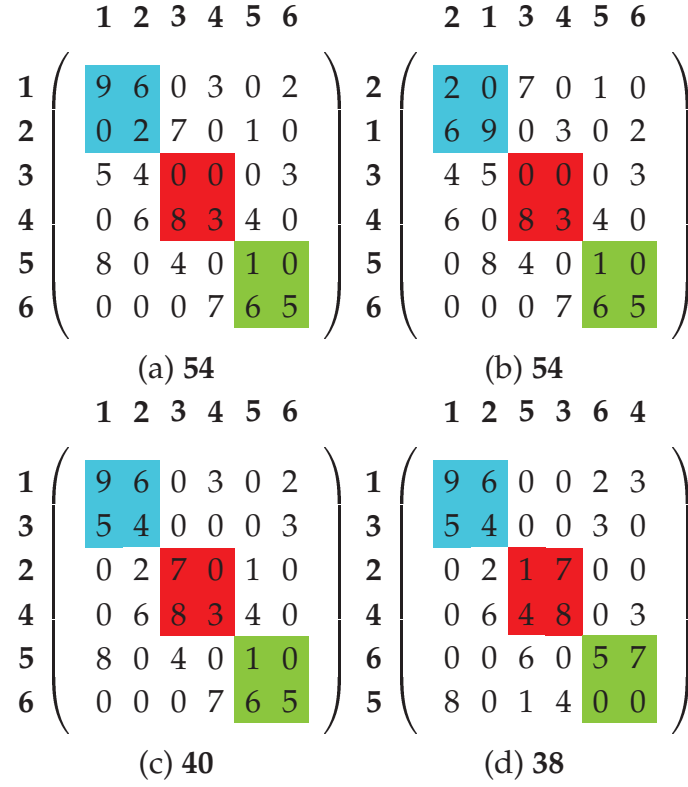
$$\chi_1(\Pi) = \sum_{(i,j) \in \mathcal{E}} |w_{ij}| \quad \text{for } i \in \mathcal{V}_{l_1}, j \in \mathcal{V}_{l_2}, \quad (2.3)$$

where  $\Pi = \{\mathcal{V}_1, \mathcal{V}_2, \dots, \mathcal{V}_K\}$  is a  $K$ -way partition of  $\mathcal{G}$ .

In Fig. 2.5, three different reordering routines are applied to the input matrix (a) with the underlying objective to find a 3-way partitioning of the adjacency matrix. It is assumed that the goal is to partition the matrix row-wise such that each partition contains two rows. The weight of the entries over all diagonal blocks must be maximized, as off-diagonal entries contribute to the edgcut. The edgcut is 54 when no partitioner is used. In Fig. 2.5(b), a spectral ordering (see Sec. 2.3.3) reduces the bandwidth of the matrix by 1, but the reordering has no effect on the edgcut. The partitioning of a 1-D partitioner (see Fig. 2.5(c)) decreases the edgcut by 14 using as row permutation  $\Pi_r = (1\ 3\ 2\ 4\ 5\ 6)$ . For this matrix, a 2-D partitioner is able to further reduce the edgcut to 38 using row permutation  $\Pi_r = (1\ 3\ 2\ 4\ 6\ 5)$  and column permutation  $\Pi_c = (1\ 2\ 5\ 3\ 6\ 4)$  (see Fig. 2.5(d)).

As minimizing Eq. 2.3 is not properly describing the communication volume for applications, e.g., in parallel sparse matrix–vector multiplication, a hypergraph-based objective function was introduced [32, 113]. A hypergraph  $\mathcal{G}_h$  can be constructed from matrix  $\mathcal{A}$  in different ways. For instance, in a *column-net* model, each row  $i$  represents a vertex  $i \in \mathcal{V}$  and each column  $j$  is a hyperedge  $e_j \in \mathcal{E}_h$ . Each hyperedge contains vertices corresponding to rows which have nonzero entries in column  $j$ . In a *row-net model*, roles of rows and columns are interchanged.

If either the row- or column-net model is applied, a 1-D partitioning is received. For applying both a row- and column-net model 2-D partitioning approaches are introduced. For instance, in a *fine-grain* 2-D hypergraph approach, each entry  $a_{ij}$  is modeled as a vertex  $v \in \mathcal{V}$ ; rows and columns are modeled as hyperedges via row- and column-net models, respectively. In a *coarse-grain* approach, a row-net model and a column-net model are applied to obtain two hypergraphs  $\mathcal{G}_{h_1}$  and  $\mathcal{G}_{h_2}$ , respectively, and both models are considered as input for the subsequent multilevel framework.



**Figure 2.5:** Comparison of partitioning methods with edgecuts (top): input matrix (a), spectral reordering (b), 1-D  $K$ -way partitioner (c), 2-D  $K$ -way partitioner (d) with  $K = 3$ .

The  $K$ -way partitioning problem in  $\mathcal{G}_h$  should also satisfy the balance constraint and minimize  $\chi_2(\Pi)$ , for example, with

$$\chi_2(\Pi) = \sum_{e_j \in \mathcal{E}_h} w_j(\lambda_j - 1), \quad (2.4)$$

where the *connectivity*  $\lambda_j$  denotes the number of parts which have vertices in the hyperedge  $e_j$ .

### 2.3.2 Multilevel Framework and Software

The  $K$ -way partitioning problem is  $\mathcal{NP}$ -hard, thus efficient heuristics are developed that partition sparse (hyper)graphs with millions of vertices. In this subsection, the notion “graph” refers to both graphs and hypergraphs.

The most widely implemented technique to partition large graphs is a multilevel scheme. The idea is to coarsen the original graph down un-

til a predetermined threshold for the size of the graph has been reached. Then, a heuristic cuts the coarse graph into the desired partitions, and the partitions are prolonged back towards the original input graph. Hence, the multilevel paradigm consists of three phases: *coarsening*, *initial partitioning*, and *uncoarsening*.

In the coarsening phase, a multilevel clustering is applied starting with the original graph by adopting greedy heuristics for maximal matching until the number of vertices in the coarsened graph falls below a predetermined threshold. The task in this step is to match similar vertices so that the small graphs capture the essential structure of the original one. Often, the quality of the entire partitioning depends on the quality of the coarsening phase. A common greedy heuristic is known as the heavy-connectivity matching or heavy-edge matching. This matching heuristic visits the vertices step by step, and matches each unmatched vertex to the neighboring unmatched one with the heaviest edge.

In the initial partitioning phase, a partition is obtained on the coarsest graph using one of various heuristics. This step is, for instance, performed by simple and fast greedy heuristics — a quite common one is known as greedy graph growing, in which a breadth-first search-like heuristic is starting from a seed vertex and terminates if a sufficiently large partition is obtained. Other approaches include bisecting the coarse graph recursively until the desired number of partitions is reached [217] or using a spectral ordering.

In the uncoarsening phase, the partition found in the second phase is successively prolonged back towards the original graph by refining the partitions on the intermediate level using one of various heuristics. Common refinement heuristics are localized iterative improvement methods which try to improve the solution by exchanging vertices among partitions [83, 130].

There are many sequential and parallel software packages available which partition the graph following the multilevel paradigm. An overview is given in Table 2.1. Based on a hypergraph model, 2-D partitioning has been successfully applied to a wide range of applications and it could be shown that its solution quality is superior to 1-D partitioning [32]. Since a large number of software products for graph partitioning exist, a DIMACS implementation challenge is devoted to this topic [1].

**Table 2.1:** Available software packages for (hyper)graph partitioning.

	sequential	parallel
<b>graph</b>	CHACO [114], METIS [126], SCOTCH [187]	JOSTLE [234], PARMETIS [128], PT-SCOTCH [53]
<b>hypergraph</b>	HMETIS [127], ML-PART [43], MONDRIAAN [229], PATOH [48]	PARKWAY [227], ZOLTAN [38]

### 2.3.3 Spectral Orderings

Another way to partition the graph is via spectral ordering algorithms. The orderings are categorized as “spectral” methods as the heuristics compute an approximation of the eigenvalues of the weighted adjacency matrix from the graph. The objective of spectral ordering algorithms is to minimize the bandwidth and to reduce the profile of a matrix. Both the bandwidth minimization and the profile reduction problems are  $\mathcal{NP}$ -complete; thus, heuristics are employed with the objective to permute nonzero entries around the diagonal. Several heuristics are implemented in software libraries.

The most traditional bandwidth and profile minimization heuristic is the reverse Cuthill–McKee (CM) method based on an unweighted adjacency matrix [60, 95]. The original method proceeds in steps. It starts the ordering with a vertex. Then, at each step the unnumbered neighbors of visited vertices are ordered in increasing order of their degrees. The reverse CM (RCM), which reverses the order found by CM is more commonly used. Choosing the starting vertex has a strong influence on the quality of the ordering [96].

There are enhanced variants of the basic ordering scheme described above with promising techniques like Sloan’s algorithm [220] and spectral reorderings. The idea in Sloan’s algorithm is to approximate the diameter of the graph step by step by choosing start and end vertices. All vertices are ranked due to metrics like the distance to the end vertex. Then, the start vertex is selected for reordering first, and all eligible vertices are chosen in an order that vertices with a higher rank are prioritized.

Spectral reorderings order the vertices according to the components of the eigenvector corresponding to the second-smallest eigenvalue of the Laplacian of a graph — the *Fiedler* vector — which minimizes the

quadratic function inherently related to the profile of the matrix and the connectivity of the graph [84].

Similarly to the multilevel scheme for graph partitioning, spectral ordering algorithms are also extended with this basic idea and integrate the computation of the Fiedler vector and Sloan's algorithm into the multilevel paradigm [21, 144]. In particular, one multilevel algorithm, called MC73, considers also the weights of the graph and returns an ordering where nonzero entries are placed around the diagonal according to the numerical value [214]. Recently, a parallel algorithm, called TRACEMIN-FIEDLER, has been designed to compute the Fiedler vector with input of weighted graphs [160].

## 2.4 Dense Subgraph Problem

Detecting a subgraph with specific constraints in graph  $\mathcal{G}$  is an objective in applications such as in community detection in social networks [85, 98], linear equation system solver [206], or genetic engineering [44]. The *weighted dense  $k$ -subgraph problem* (also called *maximum dispersion problem*) can be stated as finding a subgraph  $\mathcal{H} = (\mathcal{U}, \mathcal{F}, w)$  with  $|\mathcal{U}| = k$  and total maximum edge weight,  $\max \sum_{e \in \mathcal{F}} w(e)$ . The problem is known to be  $\mathcal{NP}$ -hard by reduction to the maximum clique problem [81]. The weighted dense subgraph problem can also be transformed into a quadratic knapsack problem and solution heuristics and lower bound computations can be adapted as well [190]. Note that if the cardinality constraint is neglected the problem can be solved in polynomial time by weighted graph matching algorithms.

### 2.4.1 Landscape of Algorithms

Solution approaches for the weighted dense  $k$ -subgraph problem are designed theoretically in the form of approximation algorithms [10, 61, 131, 139] and, practically, on the one hand, in the form of deterministic *construction heuristics* or *local improvement* methods, and, on the other hand, in the form of nondeterministic *metaheuristics* [36]. A major subfield of metaheuristics is evolutionary computing, which includes popular methods like evolutionary algorithms, tabu search, and swarm intelligence.

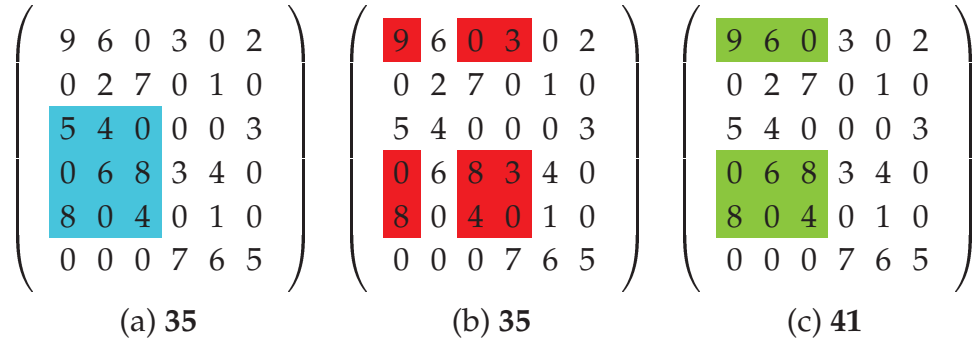
Construction heuristics iteratively improve a partial solution until  $k$  vertices have been selected. For instance, any greedy heuristic is a representative of this class. Local improvement heuristics start with a solution

of a construction heuristic and locally enhance the quality of the solution by exchanging selected vertices with unselected vertices. These heuristics are purely local search methods, and search in the neighborhood of the current solution. In order to explore not only the local neighborhood but also the global landscape of the solution space, metaheuristics provide efficient approaches.

Metaheuristics orchestrate an interaction between the local improvement heuristics and randomized solution strategies to escape from local optima and perform a robust search in the solution space. The underlying idea behind methods classified as metaheuristics is the same: an initial solution is generated randomly or by constructive heuristic, then an iterative local search procedure enhances the solution until an acceptable solution quality is achieved. Most common metaheuristics are population-based approaches like *evolutionary algorithms* or use an adaptive memory during the local search procedure like *tabu search*. Metaheuristics offer efficient techniques to solve hard combinatorial optimization problems as they are derivative-free; they offer constraint-handling techniques; they can be parallelized; they can be extended for multiobjective optimization problems; and in practice, they return a solution of high quality [5, 93, 202, 203, 233].

However, there is no guarantee of optimality of the solution. A metaheuristic might be an inappropriate technique if the evaluation function is expensive in terms of time as typically many objective function evaluations are required. The only way out of this dilemma is to substitute the original expensive function evaluation with an approximative variant which will bias the search process.

Parallel strategies for metaheuristics can be categorized into three different groups [57]. In the strategies of the first group, merely data parallelism is exploited within an iteration. This will accelerate the computation, but it does not contribute to the exploration of the solution space. The second group of strategies partitions the decision variables into smaller sets, running a heuristic on each subset while fixing the nonlocal decision variables. To obtain the final result, the partial solutions are combined again. In this manner, the solution space is traversed quite differently from a sequential run. The third group of strategies introduces diversity by starting on multiple search paths in parallel: search heuristics traverse the solution space concurrently, may communicate intermediate solutions to each other, and salvage the new information. Recently, practical implementations of these heuristics mentioned



**Figure 2.6:** Illustration of the bipartite weighted dense  $k$ -subgraph problem setting  $k = 3$  using different construction heuristics.

above have been benchmarked extensively on small-sized artificial dense graphs [9, 13, 121, 164].

In the case a bipartite graph  $\mathcal{G}_b$  is given, the problem can be reformulated as seeking a weighted bipartite subgraph  $\mathcal{H}_b = (\mathcal{U}_1, \mathcal{U}_2, \mathcal{F}, w)$  with  $|\mathcal{U}_1| = |\mathcal{U}_2| = k$  while maximizing  $w(\mathcal{F}) = \sum_{e \in \mathcal{F}} w(e)$ . Practical algorithms in the problem class are scarce [7, 222]. In Fig. 2.6, three different heuristics are used to solve the bipartite weighted dense subgraph problem for a subgraph size of 3. In Fig. 2.6(a), the heuristic finds a subgraph of weight 35 with the constraint on searching for consecutive rows and columns. In Fig. 2.6(b) the weight of an equal magnitude has been achieved by choosing nonsuccessive rows and columns, while in Fig. 2.6(c) the optimum has been detected.

As the bipartite weighted dense  $k$ -subgraph problem is  $\mathcal{NP}$ -hard, evolutionary algorithms are considered to obtain a reasonably good solution.

### 2.4.2 Evolutionary Algorithms

Evolutionary algorithms (EAs) [64, 77] are bio-inspired stochastic search heuristics, which are able to efficiently solve combinatorial optimization problems in many practical applications. An EA is a metaheuristic and involves randomization in the iterative search process. In the thesis, the traditional notion of EAs is used: it comprises the four classes of genetic algorithms, genetic programming, evolution strategies, and evolutionary programming.

Evolutionary algorithms mimic the natural selection process in a *population* with the underlying principle of *survival of the fittest*. Thus, only



**Algorithm 2.2:** *General scheme of an evolutionary algorithm.*

```
1: Initialize population
2: Evaluate individuals
3: while termination criterion not satisfied do
4:   Select parents
5:   Crossover parents
6:   Mutate resulting offsprings
7:   Evaluate offsprings
8:   Select individuals for the next generation
9: end while
```

the fittest *individuals* in a population survive after several *generations* (iterations) and represent the best solution to the problem. In each generation, new individuals, called *offsprings*, are generated by *mutation* and *crossover* of existing individuals, called *parents*. Mutation and crossover are applied randomly, as in nature, to the genes of the individuals. Mutation changes only the genes of a single individual, while crossover mixes genes of at least two individuals. The representation of the genes of an individual is problem dependent and might be coded as binary or real-valued decision variables, or even as a permutation. After an evaluation of the individuals with the *fitness function*, a *selection* process randomly picks new individuals for the next generation. Often, the fitness function is the objective function of the optimization problem. This process ends if a termination criterion is fulfilled. A typical outline of such an algorithm is given in Algorithm 2.2.

The computation in evolutionary algorithms is embarrassingly parallel. Parallelism strategies in evolutionary algorithms are either implemented by the strategies in the first and third group [5]. According to the strategy in the first group, mutation, crossover, and selection are applied in parallel on a subset of the individuals which accelerates the computation itself using data parallelism. By allowing a generation wide parallelization, the strategy of the third group can be further split into two categories: *coarse-grained* and *fine-grained* parallelization.

In the coarse-grained parallelization scheme, the entire population set is divided into  $P$  parts so that each process obtains one part, performs the same local evolutionary algorithm, and exchanges individuals among subpopulations with a *migration* operator. In a *stepping-stone* model indi-



viduals are migrating only to neighboring processes, while in the *island model* individuals are not restricted to certain processes.

In the fine-grained parallelization scheme, ideally, a single individual is assigned to a process. The process forms a subpopulation by coalescing with neighboring processes and runs an EA in this new subpopulation. Subpopulations are overlapping with neighboring subpopulations; thus, information in a subpopulation is also circulating as in the stepping-stone model of the coarse-grained approach.



## Chapter 3

---

# Auctions in Bipartite Graph Matching

---

In this chapter, the principle of auctions in matching algorithms is presented to solve weighted bipartite graph matching problems efficiently. The auction mechanism has its origin in game theory. Game theory is a research field highly relevant in practice; it deals with the behavior of persons based upon the choices of other persons. Till today, eight Nobel memorial prizes in economic sciences have been awarded within this research field. Auction theory is an application area of game theory and is of fundamental relevance for practical decision making in business and economy. In the classical sense of an auction, an auctioneer opens the auction, bidders submit bids for an object of interest, and the single bidder with the highest bid obtains the object. Auctions are often used in scenarios in which buyers do not know the maximum bids of other buyers or in which sellers do not know the maximum bids of the buyers. In the context of the bipartite graph matching, auction algorithms simulate an auction as a deterministic iterative optimization process.

### 3.1 Auction Theory

Auction mechanisms have entered daily life, in particular, in web applications like *eBay* (<http://www.ebay.com>) or *PsychoAuction* (<http://www.psychoauction.com>). Auctions are set up quite differently in these two application examples.

At *eBay*, an object is offered to bidders with an initial starting price defined by the seller. Bidders judge the object due to their own value function and place bids for the object asynchronously in the online system. However, a feasible bid must be higher than the current price of the object, as the price for the object is updated by a fixed increment. Thus, the actual bid is not disclosed, but only the new price. At the end of the auction procedure — defined by predetermined date and time, which is visible to all — the bidder with the highest offer pays the last price for the object to the seller. The online system earns money by receiving fees from the seller.

In *PsychoAuction*, bidders buy a virtual concurrency, called *pennys*, beforehand, which is also the incremental value for the price. Objects are offered with a small initial price set by the online system itself. Now, buyers asynchronously bid a penny for an object. However, they are only permitted to bid for the same object a number of times. Each offer resets a countdown timer to a small value which will determine the termination process. The online system earns money by selling more pennies than the market price of the object.

There are many types of auctions and variations thereof. Four general types can be identified: *ascending-bid*, *descending-bid*, *first-price sealed-bid*, and *second-price sealed-bid*. The most common type are ascending-bid auctions, which raise the price for an object gradually until the bidder with the highest bid remains. For instance, auctions in selling art work or *eBay* and *PsychoAuction* are ascending-bid auctions. In contrast, descending-bid auctions start with a high initial price and decrease it until a bidder is willing to pay the price. For instance, televised home shopping programs often use this method. In the last two auction principles, first-price and second-price sealed bid auctions, bids — historically sealed in an envelope — are submitted concurrently, and the highest bid wins the competition. In the former strategy, the highest bidder pays the submitted bid; in the latter one, the bidder pays only the price of the second highest bid.

These types of auction principles have been analyzed extensively for matching markets [74, 133] where interaction between buyers and sellers are modeled as bipartite graphs with disjoint sets of buyers and objects. Each buyer has an associated *benefit* for an object and pays a *price* to be the new owner of the object. The *profit* which a buyer achieves is defined as the benefit minus the current price of an object. A common objective in matching markets is to converge to *market clearing prices* where every buyer is assigned to an object and achieves the maximum price-benefit

ratio, i.e., a perfect weighted matching in a bipartite graph.

In matching market analysis, it has been shown that buyers should update the price of the object with the bid computed by the highest profit minus the second-highest profit. Hence, the buyer is *indifferent* between the best and second-best object as the profit of the best object, defined by the benefit minus the updated price is equal to the profit of the second-best object. If other buyers overbid the buyer it is worthwhile for the buyer to bid for the second-best object in the next iteration due to an optimal price-benefit ratio. This process will conclude with the assignment of market clearing prices where every buyer is satisfied with the obtained object.

## 3.2 Sequential Auction Algorithms

Auction algorithms [29] find weighted matchings via the game-theoretic idea of an ascending-bid auction. The mapping of the bipartite graph to the auction process is performed as follows: vertices of  $\mathcal{V}_1$  and  $\mathcal{V}_2$  represent buyers and objects, respectively, and weighted edges symbolize the benefits. The auction-based algorithm (see Algorithm 3.1) consists of three phases: the *initialization phase* (lines 1–4), the *bidding phase* (lines 6–11), and the *assignment phase* (lines 12–13). Each object  $j$  has an associated price  $p_j$ , which is initially set to zero. In the bidding phase in the method `choose(I)`, an unassigned buyer  $i$  is chosen from the set of unassigned buyers  $I$ , here in a cyclic order. At the beginning, a buyer with the smallest index in the set is selected, followed by the second-smallest index in the next auction iteration, until the largest index entry has been reached. Then, the procedure is repeated with an unassigned buyer at the smallest index in the updated set.

Each buyer bids for the object  $j_i$ , where object  $j$  has maximal profit for buyer  $i$ . If such an object does not exist, which is checked in line 9, there is no profitable object available and the buyer will remain unmatched (line 16). Otherwise, the highest profit  $u_i$  for buyer  $i$  is computed by  $w_{ij_i} - p_{j_i}$ , while the second-highest profit  $v_i$  is calculated by ignoring the most valuable object  $j_i$ . The bid is computed by subtracting the second-highest profit  $v_i$  from the highest profit  $u_i$ , i.e.,  $u_i - v_i$ . After the unassigned buyer has submitted the bid, the designated object is assigned to the bidder. The new price is calculated by increasing the old price by the corresponding bid and by a small increment  $\epsilon$ . The role of  $\epsilon$  is discussed

**Algorithm 3.1:** *Sequential Auction Algorithm for Weighted Matching*

**Input:** Bipartite graph  $\mathcal{G}_b = (\mathcal{V}_1, \mathcal{V}_2, \mathcal{E}, w)$   
**Output:** Matching  $\mathcal{M}$

```

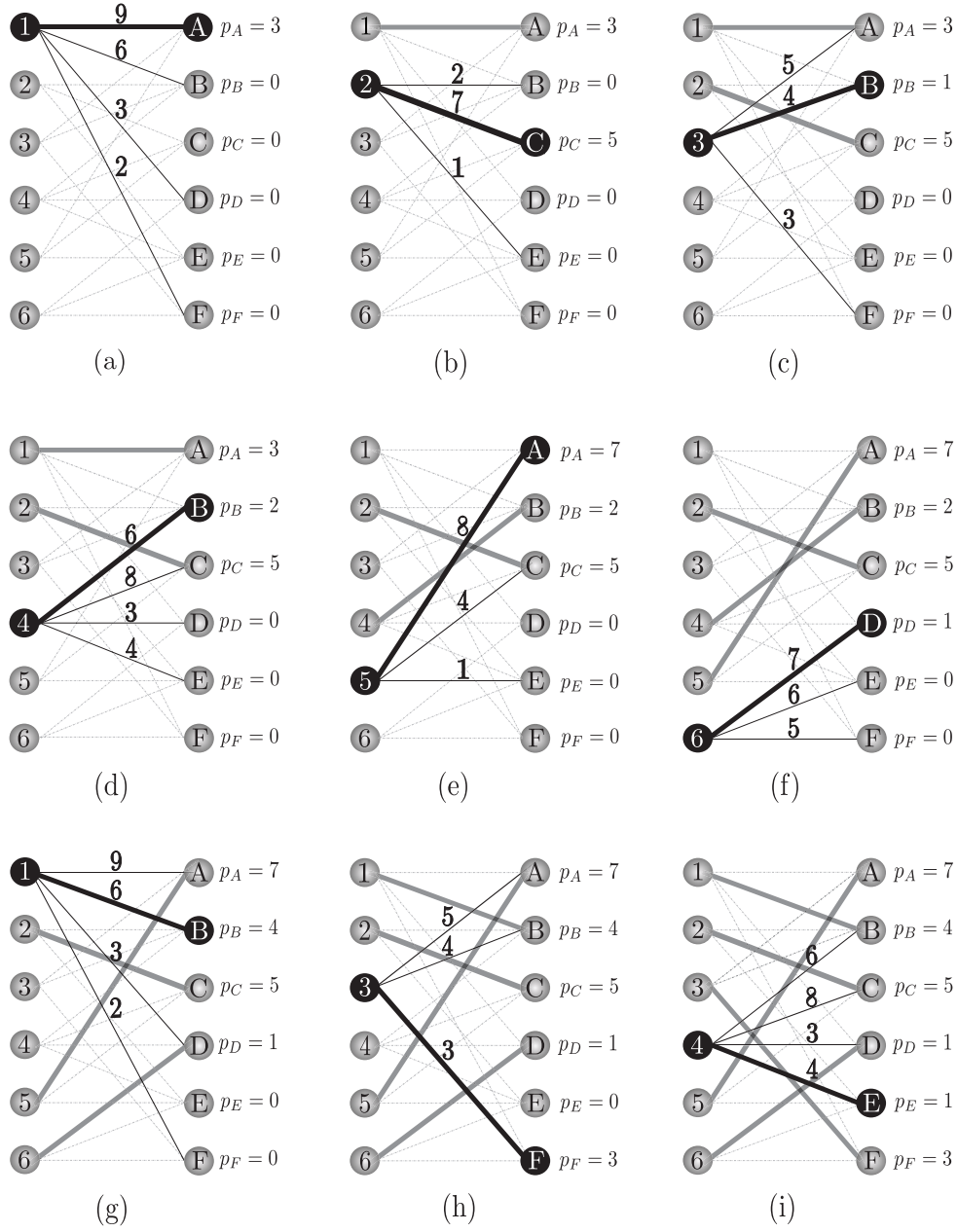
1:  $M \leftarrow \emptyset$  ▷ current matching
2:  $I \leftarrow \{i : 1 \leq i \leq n_1\}$  ▷ set of unassigned buyers
3:  $p_j \leftarrow 0$  for  $j = 1, \dots, n_2$  ▷ initialize prices for objects
4: initialize( $\epsilon$ ) ▷ discussed in Sec. 4.2
5: while  $I \neq \emptyset$  do ▷ auction iteration
6:    $i \leftarrow \text{choose}(I)$  ▷ determine a free buyer
7:    $j_i \leftarrow \arg \max_j \{w_{ij} - p_j\}$  ▷ find the best object for buyer  $i$ 
8:    $u_i \leftarrow w_{ij_i} - p_{j_i}$  ▷ store profit of the most valuable object
9:   if  $u_i > 0$  then
10:     $v_i \leftarrow \max_{j \neq j_i} \{w_{ij} - p_j\}$  ▷ store second-best profit
11:     $p_{j_i} \leftarrow p_{j_i} + u_i - v_i + \epsilon$  ▷ update price with bid  $u_i - v_i$  and  $\epsilon$ 
12:     $\mathcal{M} \leftarrow \mathcal{M} \cup \{i, j_i\}; I \leftarrow I \setminus \{i\}$  ▷ assign buyer to object
13:     $\mathcal{M} \leftarrow \mathcal{M} \setminus \{k, j_i\}; I \leftarrow I \cup \{k\}$  ▷ free previous owner  $k$ 
14:    update( $\epsilon$ ) ▷ discussed in Sec. 4.2
15:   else
16:      $I \leftarrow I \setminus \{i\}$  ▷ no profitable object available for buyer  $i$ 
17:   end if
18: end while

```

in Sec. 4.2. The next iteration is started with an unassigned buyer. This process is repeated until every buyer has been matched to an object. During the algorithm the number of assigned buyers is never decreasing, but an assigned buyer may get unassigned again.

The execution of the sequential auction algorithm that determines the perfect weighted matchings is explained by mapping the  $6 \times 6$  sparse adjacency matrix  $\mathcal{A} = (a_{ij})$  to the bipartite graph representation as illustrated in Fig. 3.1. The dotted light gray lines show the available edges. The solid lines with weights indicate the candidates for the buyer. The bold gray lines illustrate the matchings of the previous iterations, and the bold black lines present the matching of the current iteration. For illustration purposes,  $\epsilon$  is set to 0.

The buyers  $1, 2, \dots, 6$  are competing for the objects  $A, B, \dots, F$ . The prices of the objects  $p_A, p_B, \dots, p_F$  are initialized to zero and updated after each iteration. In Fig. 3.1(a), buyer 1 starts by finding the most valuable object among  $A, B, D$ , and  $F$ . The most profitable object is  $A$  with



**Figure 3.1:** Nine iterations of the sequential auction algorithm for a sparse maximum weight matching problem with six buyers  $1, 2, \dots, 6$  and six desired objects  $A, B, \dots, F$  with  $\varepsilon = 0$ .

a value of  $w_{1A} - p_A = 9 - 0 = 9$ . The second most profitable object for buyer 1 is object B with a value of  $w_{1B} - p_B = 6$ . Thus, buyer 1 selects object A and increments the price  $p_A$  of the object by 3. In the second iteration, Fig. 3.1(b), buyer 2 acquires object C with a similar argument

and the price of the object is updated accordingly, while the matching between buyer 1 and object  $A$  still remains. The next iteration, Fig. 3.1(c), shows the influence of the previous matchings on buyer 3. Object  $A$  is the most valuable object for buyer 3, but the best profit is returned by object  $B$  with a value of 4. Thus, the edge  $(3, B)$  is added to the matching while  $p_B$  is incremented by one. Buyer 4 in Fig. 3.1(d) retrieves the highest profit of 5 for object  $B$  and the second-highest profit for object  $E$  with a value of 4. The price for object  $B$  is further incremented by one, buyer 3 gets unassigned, and buyer 4 will now be the new owner of object  $B$ .

The next five iterations, Figs. 3.1(e)–3.1(i), illustrate the auction procedure until all buyers are satisfied with the purchased objects, i.e., each buyer is incident to exactly one matching edge.

It may happen that buyers gain benefit by only one object, i.e., vertices in  $\mathcal{V}_1$  with a degree of one. Assuming a perfect weighted matching is the overall objective, buyers with only one incident edge are definitely assigned to their corresponding objects in the matching. Therefore, in the first auction iteration, these buyers set the bid for the objects to  $C = \max_{ij} \{|w_{ij}|\}$  and the prices are updated by this high value. This prevents other buyers from bidding for these very expensive objects in the subsequent iterations. Note that this technique is quite similar to the principle applied in the Karp–Sipser heuristic, as vertices with a degree of one are matched first [147].

This type of auction algorithm is called a *forward auction algorithm*. If the roles of buyers and objects are interchanged, a *reverse auction algorithm* is attained, in which objects compete for buyers [47]. But the focus of this thesis is purely on forward auction algorithms.

### 3.3 Existing Parallel Auction Algorithms

Since the goal is to develop and design a scalable parallel auction algorithm for massively parallel architectures, existing parallelization strategies are reviewed.

On shared memory architectures, *synchronous* and *asynchronous* variants of the auction algorithm have been implemented and compared to each other in terms of efficiency and speed [28]. In the asynchronous approach, the bid computation is typically based on out-of-date price information. Thus, the bid for an object does not always correspond to the current price of an object. The advantage of the asynchronous implemen-



tation is the speedup in the bid computation while still converging to a maximum matching with high weight. However, the number of auction iterations may increase due to outdated bid computation. In contrast, the synchronous implementation separates the bidding and assignment phase using synchronization points.

For distributed memory architectures, there have been two attempts to develop a parallel auction algorithm for dense and sparse graphs [42, 196].

In the case of dense graphs, an algorithmic technique called *look-back bidding* has been added to the auction algorithm [42]. This technique stores previous biddings of each buyer in a working set and computes new bids by using the information in the set. Methods are also introduced to maintain the set, which can accelerate the bid computation. The algorithm outperforms existing forward and forward-reverse implementations. However, the algorithm does not achieve any reasonable speedup because the algorithm iterates the same way as the sequential implementation.

For sparse bipartite graphs, a distributed matching algorithm was developed that finds a maximal weighted matching [196]. The idea of the algorithm is that a block of the bipartite graph is assigned to a process and each process is responsible for free buyers in its block. A *blocked local auction* for free buyers which contains the bidding phase and the price update is performed. Then, modified prices are collected and synchronized before the next local auction iteration starts. The algorithm has been tested on sparse bipartite graphs and some speedup could be achieved by using a small number of processes.

The focus of this thesis is to develop distributed auction-based matching algorithms that are performance scalable on massively parallel architectures. The key concept is to use different  $\varepsilon$ -scaling strategies in order to find weighted matchings in sparse and dense and in balanced and unbalanced bipartite graphs. It is implicitly assumed that the graph is already distributed among all compute nodes using a  $\mathcal{V}_1$ -wise distribution. Note that other advanced graph partitioning methods can be also applied to the graph before starting the auction process (see Sec. 2.3).



## **Part II**

# **Parallel Graph Matching**



## Chapter 4

---

# Design of Parallel Auction Algorithms

---

Auction-based weighted matching implementations exhibit a high potential for solving large graph matching problems on massively parallel architectures due to the fact that the computation part, i.e., the bidding phase, is embarrassingly parallel. Recall that the building blocks of an auction algorithm are the bidding and assignment phases. The bidding phase contains the bid computation of a free buyer, which can be performed independently of other free buyers. The assignment phase comprises matching of a buyer to an object correlated with its price update, where in a distributed version only the bidder with the highest bid for an object is considered for the assignments. Since the target data structures are bipartite graphs, the notation of such a graph is reviewed again. A bipartite graph  $\mathcal{G}_b = (\mathcal{V}_1, \mathcal{V}_2, \mathcal{E}, w)$  consists of the vertex sets  $\mathcal{V}_1, \mathcal{V}_2$ ,  $|\mathcal{V}_1| = n_1$ ,  $|\mathcal{V}_2| = n_2$ ,  $n_1 \leq n_2 = n \in \mathbb{N}$ , edge set  $\mathcal{E} \subseteq \mathcal{V}_1 \times \mathcal{V}_2$ , weight function  $w : \mathcal{E} \rightarrow \mathbb{R}$ , and  $w_{ij} := w(i, j)$ .

### 4.1 PAUL — A Parallel Auction Algorithm Implementation

Algorithm 3.1 consists of the three phases: initialization, bidding, and assignment. The bidding phase contains the bid computation of a free buyer, and the assignment phase includes the matching of the buyer to

the object and the price update of the object. In a parallel version of the algorithm, contrary to the sequential version, the bids of free buyers can be computed in a single auction iteration. Each free buyer computes a bid for the most valuable object according to the current price of the object. Then, the buyer with the highest bid for an object is determined and is assigned to the object. The prices of the objects are updated according to the highest bids. Then, the parallel bidding phase starts again with the free buyers. These phases can be interpreted as the supersteps *computation* and *communication* in the bulk synchronous parallel (BSP) programming model [31]. In a BSP algorithm, sequences of the supersteps are executed; the superstep “computation” represents a bidding phase and the superstep “communication” corresponds to the assignment phase. Synchronization points between the supersteps separate the phases from each other. However, in contrast to the BSP algorithm, the parallel auction algorithm implementation PAUL does not rely on global synchronization points and explicit barriers are replaced by point-to-point synchronization mechanisms.

The only communication part in PAUL is the exchange of local prices for the objects among the processes so that the winner of the object can be found. The communication size is reduced by exchanging only locally altered prices, and by bundling messages into one single message. Additionally, when local buyers on a process are interested in the same object, the highest price is ascertained locally among all the prices for the same object before submitting the new prices to all other processes.

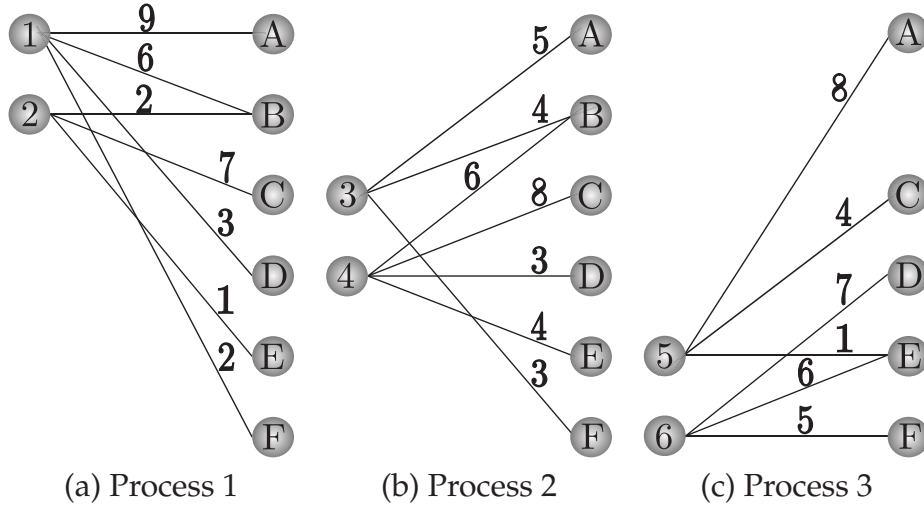
The parallel auction-based algorithm (see Algorithm 4.1) also consists of the three phases of Algorithm 3.1: the *initialization phase* (lines 2–7), the *bidding phase* (lines 9–17), and the *assignment phase* (lines 18–23) [204]. The term *local* is attached to variables to indicate the fact that the variable is only visible to the owner process, and the notion *global* refers to globally visible, synchronized variables. In the preprocessing step, the bipartite graph is distributed among the processes (line 1). Each process initializes the data structures as in the sequential algorithm and computes the bids for free buyers in its block (lines 2–17). The unassigned buyers are processed in a cyclic order as presented in the sequential auction algorithm. However, in contrast to the sequential auction algorithm, not only a single buyer computes a bid, but all free buyers compute a bid based on the current prices of the objects in a single auction iteration. Then, the new owner of an object is determined by exchanging the local prices, and a buyer becomes the owner of an object if the bid is the

**Algorithm 4.1:** *Parallel Auction Algorithm (PAA) for Weighted Matchings*

**Input:** Bipartite graph  $\mathcal{G}_b = (\mathcal{V}_1, \mathcal{V}_2, \mathcal{E}, w)$   
**Output:** Matching  $\mathcal{M}_{\text{global}}$

- 1: Distribute graph  $\mathcal{V}_1$ -wise
- 2:  $\mathcal{M}_{\text{global}} \leftarrow \emptyset$   $\triangleright$  initialize global matching
- 3:  $\mathcal{M}_{\text{local}} \leftarrow \emptyset$   $\triangleright$  set of locally matched buyers
- 4:  $I_{\text{local}} \leftarrow \{i : 1 \leq i \leq \frac{n_1}{P}\}$   $\triangleright$  reindexing set of locally free buyers
- 5:  $I_{\text{global}} \leftarrow \text{allgather}(I_{\text{local}})$   $\triangleright$  globally free buyers
- 6:  $p_j \leftarrow 0$  for  $j = 1, \dots, n_2$   $\triangleright$  local price vector for the objects
- 7:  $\text{initialize}(\varepsilon_{\text{local}})$   $\triangleright$  discussed in Sec. 4.2
- 8: **while**  $I_{\text{global}} \neq \emptyset$  **do**
- 9:   **for all**  $i \in I_{\text{local}}$  **do**
- 10:      $i \leftarrow \text{choose}(I_{\text{local}})$   $\triangleright$  determine a locally free buyer
- 11:      $j_i \leftarrow \arg \max_j \{w_{ij} - p_j\}$
- 12:      $u_i \leftarrow w_{ij_i} - p_{j_i}$
- 13:     **if**  $u_i > 0$  **then**
- 14:        $v_i \leftarrow \max_{j \neq j_i} \{w_{ij} - p_j\}$
- 15:        $p_{j_i} \leftarrow p_{j_i} + u_i - v_i + \varepsilon_{\text{local}}$   $\triangleright$  update prices
- 16:     **end if**
- 17:   **end for**
- 18:    $\text{gather\_changed\_prices}()$   $\triangleright$  communication phase
- 19:   **for all**  $i \in I_{\text{local}}$  **do**
- 20:      $\text{check\_winner}(i)$   $\triangleright$  update local price
- 21:   **end for**
- 22:    $\text{update\_local\_sets}(\mathcal{M}_{\text{local}}, I_{\text{local}})$   $\triangleright$  update sets
- 23:    $I_{\text{global}} \leftarrow \text{allgather}(I_{\text{local}})$   $\triangleright$  update global free buyers
- 24:    $\text{update}(\varepsilon_{\text{local}})$   $\triangleright$  discussed in Sec. 4.2
- 25: **end while**
- 26:  $\mathcal{M}_{\text{global}} \leftarrow \text{gather}(\mathcal{M}_{\text{local}})$   $\triangleright$  collect the final matching

globally highest bid in the iteration (lines 18–23). In the method  $\text{gather\_changed\_prices}()$ , each process stores altered prices in a local list and gathers the list among all the processes. In the method  $\text{check\_winner}(i)$ , each process updates the local prices of the objects according to the lists, and updates either the sets of unassigned buyers or the local matching set. If there is no valuable object for buyer  $i$  available, which is checked in line 13, then the buyer  $i$  is removed from the set of unassigned buyers. The next iteration starts if there are still free buyers in a process. When the set of unassigned buyers is empty, the global matching is collected (see



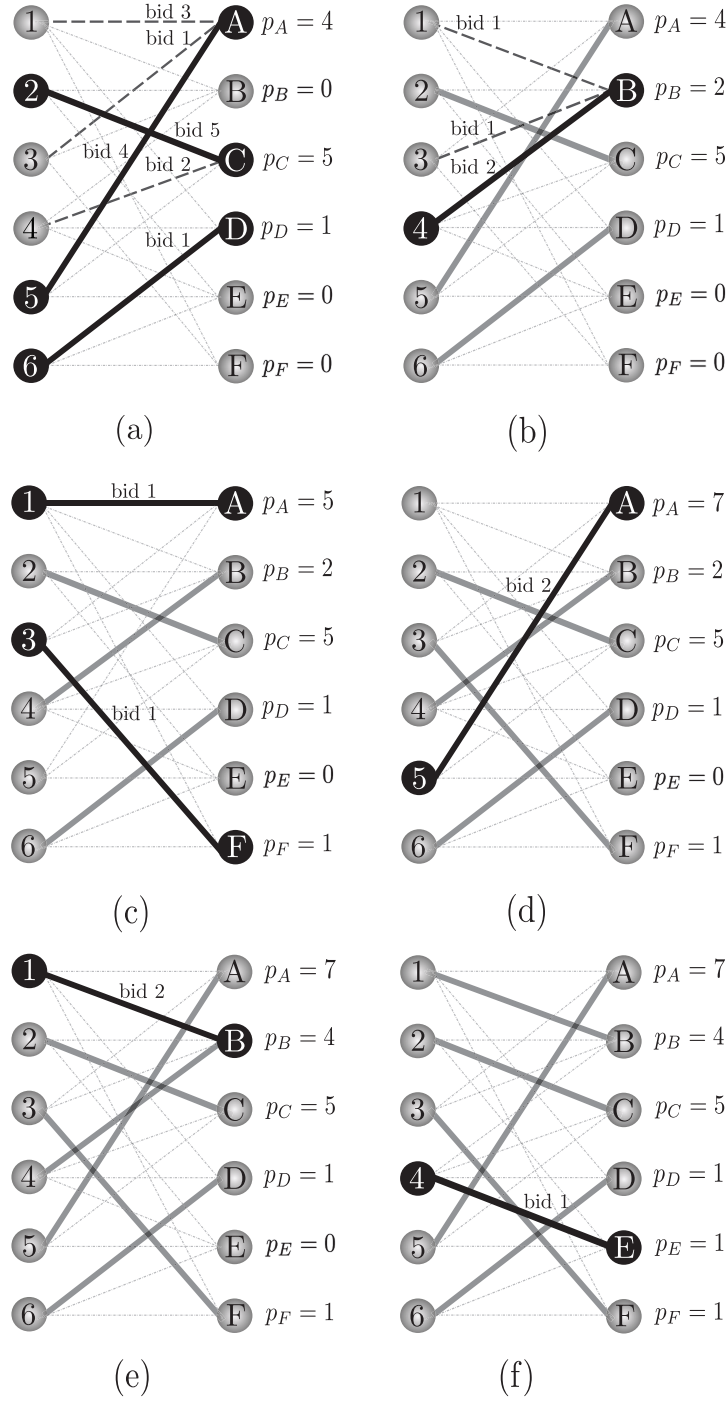
**Figure 4.1:**  $\mathcal{V}_1$ -wise distribution of the sparse balanced bipartite graph.

line 26). Processes with no unassigned buyers are only acting in the price update and are reactivated if another buyer has set the current highest price for the object. The  $\varepsilon$ -scaling strategy is discussed in Sec. 4.2. Note that the entire algorithm runs in a fully parallel distributed mode when the matrix is already distributed among the processes.

The behavior of the parallel auction algorithm (PAA) is illustrated by using three processes for the sparse  $6 \times 6$  weighted adjacency matrix (see Fig. 2.4). The graph is distributed in a  $\mathcal{V}_1$ -wise fashion (see Fig. 4.1). Two vertices of  $\mathcal{V}_1$  and the incident edges and adjacent vertices of  $\mathcal{V}_2$  are assigned to each process. In the parallel version of the auction algorithm the processes compute the best bid for each buyer in a single auction iteration. The bid is computed by the formula  $u_i - v_i$  and indicates the current optimal bid score for buyer  $i$  to get object  $j_i$  (see Fig. 4.2).

Due to the initialization of the prices of the objects to zero, in the first iteration, Fig. 4.2(a), the most valuable object can be computed solely by the two highest values of the incident edges; e.g., buyer 1 submits a bid of  $(w_{1A} - p_A) - (w_{1B} - p_B) = (9 - 0) - (6 - 0) = 3$  for object  $A$ . A label is attached above the edge to indicate the bid; here, for instance, the label is “bid 3.” Thus, object  $A$  receives three bids of value 3, 1, and 4 from buyers 1, 3, and 5, respectively. Buyer 5 submits the highest bid and is matched with object  $A$ . Objects  $C$  and  $D$  are matched with buyer 2 and buyer 6, respectively. In total, three feasible matchings are computed and the prices are updated according to the bids and synchronized among the processes. Process 2 is now idle—it starts acting again if one of its local





**Figure 4.2:** Six iterations of the PAA for the sparse maximum weighted matching problem with six buyers  $1, 2, \dots, 6$  and six desired objects  $A, B, \dots, F$  on three processes with  $\varepsilon = 0$ .

buyers gets unassigned again. In the next iteration (see Fig. 4.2(b)), the free buyers 1, 3, and 4 perform the bidding phase but only process 1 with buyer 4 makes a successful offer. The next iteration, Fig. 4.2(c), shows the procedure for the last two buyers 1 and 3. Buyer 3 matches to object  $F$  whereas buyer 1 overbids buyer 5 for object  $A$ . In iteration Fig. 4.2(d), the most valuable object for buyer 5 is object  $A$  which is assigned to the buyer. Then, Fig. 4.2(e), buyer 1 attains object  $B$  and, finally, the algorithm terminates with the last assignment of buyer 4 to object  $E$ . Here, the optimal solution of the problem is returned after six iterations.

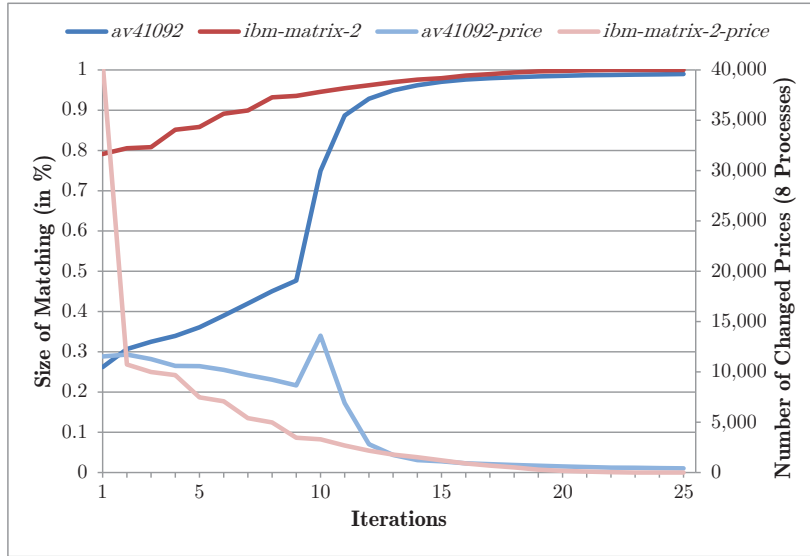
### 4.1.1 Performance Aspects

The bidding phase — i.e., finding the best and second-best profits for local free buyers — is performed on every process simultaneously. In contrast to the sequential run, every local buyer computes a bid for the most-valuable object. If local buyers are interested in the same object, only the buyer with the highest bid locally updates the price of the object. The bidding phase is accelerated using threading parallelization. Thus, the algorithm is parallelized in a two-way fashion. In the outer parallelization scheme, free buyers are computing optimal bids in the bidding phase concurrently, and in the inner parallelization scheme the bid computation itself is parallelized for each free buyer.

In the assignment phase, the highest global price of the object among all buyers is determined, and the local prices are exchanged. Every process receives a globally changed price list and checks if the price in the list is greater than the local price. If this is the case, the local price is updated and, if the buyer is locally assigned to the object, the buyer is unassigned again. The buyer with the highest price is the new owner of the object. If two buyers of different processes are bidding for the same object with an identical price, the object is assigned to the buyer who comes first in the changed price list.

The price exchange plays a distinctive role for performance of the full algorithm. To minimize the communication, only changed prices are exchanged. The prices are coalesced into a single message which is sent to the other processes. Every process obtains a changed price list and checks if buyers residing on other processes have overbid locally assigned buyers. In the implementation, a price vector  $p \in \mathbb{R}^{n_2}$  is stored on each process.

In Fig. 4.3, the typical convergence behavior of the PAA is visualized



**Figure 4.3:** Number of matched edges and communication in the parallel algorithm for two exemplary instances.

using 8 processes when looking at the first 25 iterations. On the left  $y$ -axis is the size of the matching in percent, while on the right  $y$ -axis is the number of the prices of objects that have changed from the last to the current iteration. The value of the matching size of the exemplary matrices *av41092* and *ibm-matrix-2* (see Sec. 10.3) is visualized on the left  $y$ -axis. On the right  $y$ -axis is the number of changed prices illustrated for the matrices with the attached keyword “-price.” Most progress concerning the matching size happens in the first few iterations. Then, only a small number of buyers are unassigned and they compete for the free objects. The size of the list with the number of price exchanges reduces significantly within the first 15 iterations.

## 4.2 $\varepsilon$ -Scaling Mechanisms

The strategies for choosing the initial  $\varepsilon$  and how to update  $\varepsilon$ , called  $\varepsilon$ -scaling, are very important for the optimality properties of the algorithm. However, the  $\varepsilon$ -strategies heavily depend on the edge weights of the graph. Therefore, some scaling mechanism is applied to the entire graph. The necessity of scaling of the weights is also supported by the fact that the PAA requires nonnegative edge weights  $w_{ij} \in \mathbb{R}^+$  and a formulation of the matching problem as a maximization problem.

### 4.2.1 Normalized Edge Weights

In [178] the following scaling method has been developed for a nonsingular weighted adjacency matrix  $\mathcal{A} = (a_{ij})$  to get a new matrix  $C = (c_{ij})$  which can be used to solve the minimum weighted matching problem:

$$c_{ij} := \begin{cases} \log \bar{a}_i - \log |a_{ij}| & \text{if } a_{ij} \neq 0, \\ +\infty & \text{otherwise,} \end{cases} \quad (4.1)$$

where  $\bar{a}_i := \max_{j=1, \dots, n} |a_{ij}| \neq 0$  for all  $i = 1, \dots, n$ .

The auction algorithm is designed to deal with maximization problems. Therefore, a similar scaling idea is applied to the graph:

$$c_{ij} = \begin{cases} \log |a_{ij}| - \log \bar{a}_i + \log a_{\max} - \log a_{\min} & \text{if } a_{ij} \neq 0, \\ -\infty & \text{otherwise,} \end{cases} \quad (4.2)$$

where the difference of  $a_{\max} := \max_{ij} |a_{ij}| \neq 0$  and  $a_{\min} := \min_{ij} |a_{ij}| \neq 0$  shifts the nonpositive values  $\log |a_{ij}| - \log \bar{a}_i$  to a value greater than zero.

Note that in this thesis two different objective functions are maximized. In applications dealing with sparse graphs, the scaling mechanism (4.2) must be applied, and thus  $\prod_{(i,j) \in M} |w_{ij}|$  is maximized. In the applications dealing with dense bipartite graphs, the edge weights in the graphs are already scaled to values in  $[0, 1]$ . Here, the objective function  $\sum_{(i,j) \in M} |w_{ij}|$  is maximized.

### 4.2.2 $\varepsilon$ -Scaling Strategies

Based on this initial scaling of the edge weights, various  $\varepsilon$ -scaling strategies can be applied. Using  $\varepsilon$ -scaling is a central aspect of the auction algorithms described in Algorithms 3.1 and 4.1. For instance, consider line 11 in Algorithm 3.1. In this line, a new price for the object is computed by adding the bid and the small increment  $\varepsilon$  to the old value of the price. To understand the importance of  $\varepsilon$  in the price update, assume that  $\varepsilon$  is set to zero. Additionally, imagine that two buyers are bargaining for the same valuable object while the highest and second-highest profits are of the same value. Then, the updated price remains unchanged. In such a scenario, each buyer will never be satisfied with the current assignment and the process will end in a price war, in which a small number of buyers are competing for the roughly equally desirable objects. In order to ensure that the price for an object is raised after each iteration, the small increment  $\varepsilon$  is introduced.

**Algorithm 4.2:** *The Adaptive Parallel Auction Algorithm:  $\epsilon$ -PAA*

```

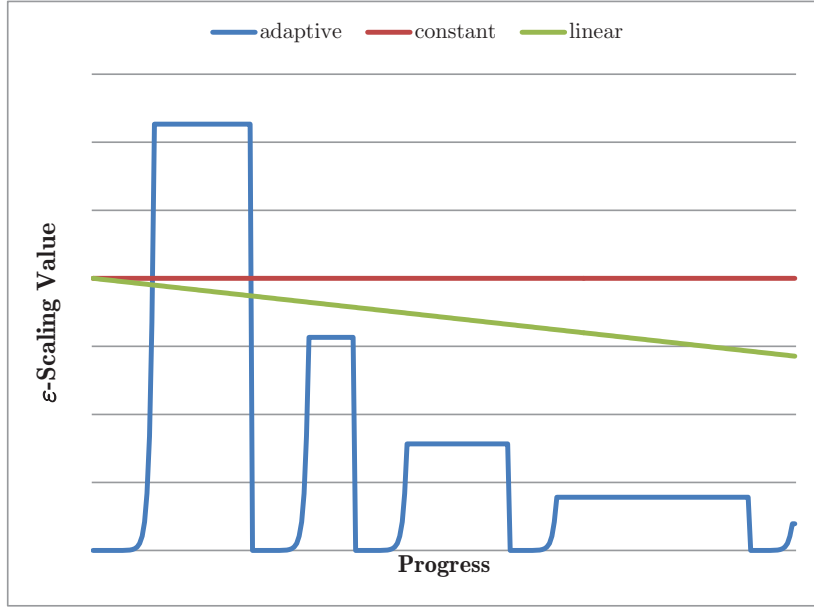
1: Perform the initialization phase of algorithm 4.1 (lines 1–6)
2:  $\xi \leftarrow 2; \theta \leftarrow 16; \gamma \leftarrow \frac{n+1}{\theta}$ 
3:  $\delta \leftarrow \left\lfloor \min \left\{ \frac{|I_{\text{global}}|}{\xi}, \frac{n}{\theta} \right\} \right\rfloor$   $\triangleright$  initialize threshold  $\delta$ 
4: while  $I_{\text{global}} \neq \emptyset$  do
5:    $\epsilon_{\text{local}} \leftarrow \frac{\theta}{n+1}$   $\triangleright$  reset  $\epsilon_{\text{local}}$  to small value
6:   while  $|I_{\text{global}}| > \delta$  do
7:     Perform bidding and assignment phase of algorithm 4.1
8:     if  $\gamma > \epsilon_{\text{local}}$  then
9:        $\epsilon_{\text{local}} \leftarrow \epsilon_{\text{local}} \cdot \xi$ 
10:    else
11:       $\epsilon_{\text{local}} \leftarrow \gamma$ 
12:    end if
13:     $\gamma \leftarrow \gamma / \xi$ 
14:  end while
15:   $\delta \leftarrow \delta / \xi; \theta \leftarrow \theta \cdot \xi$   $\triangleright$  update  $\delta$  and  $\theta$ 
16: end while

```

The initial value of  $\epsilon$  has a high computational impact on the auction algorithm; the number of iterations is almost proportional to  $\frac{C}{\epsilon}$  [27]. Ideally, the value of  $\epsilon$  should be very close to the optimal value of the price, as the number of iterations to find a matching will then be small. Thus, the idea of  $\epsilon$ -scaling is to initialize the prices by choosing a large value for  $\epsilon$  and either successively reducing the value until  $\epsilon < \frac{1}{n}$  or using the same  $\epsilon$  value throughout all auction iterations until the algorithm has converged.

However, if  $\epsilon$  is chosen too large it may happen that a maximal matching with a small weight will be received. If  $\epsilon$  is chosen too small it may happen that the algorithm will converge very slowly. Thus, it is essential to find a trade-off between quality and speed. The following initialization and update rules are typically implemented [27]:  $\epsilon$  is initialized with  $\epsilon = \frac{C}{\theta}$  and updated by  $\epsilon = \max \left\{ \frac{1}{n}, \frac{\epsilon}{\theta^t} \right\}$  in iteration  $t \geq 0$ , where  $C = \max_{ij} |w_{ij}| > 0, \theta > 0$ .

For the parallel auction-based matching algorithm, the following  $\epsilon$ -scaling strategies for sparse balanced graphs and for dense unbalanced graphs are proposed. In sparse graphs,  $\epsilon_{\text{local}}$  is initialized with  $\epsilon_{\text{local}} = \frac{n+1}{\theta}$  and decreased by  $\max\{\epsilon, \epsilon_{\text{local}} - \epsilon\}$ , where  $\theta = 16$  and  $\epsilon = \frac{1}{n+1}$ .



**Figure 4.4:** Behavior of the different  $\epsilon$ -scaling strategies over the number of iterations.

Unfortunately, in the dense case, the parallel algorithm often runs into a price war scenario and the number of iterations may dramatically rise. Consequently, the algorithm does not scale well. Hence, two other scaling strategies are developed to overcome the issues.

The first one initializes  $\epsilon_{\text{local}} = \frac{\theta}{n+1}$  and increments  $\epsilon_{\text{local}} = \epsilon_{\text{local}} + \epsilon$ , where  $\epsilon = \frac{1}{n+1}$ . This strategy also provides a maximum weighted matching.

As a second option, a heuristic was developed in which the  $\epsilon_{\text{local}}$  value is initialized with a small value and adaptively increased relative to the overall progress (see Algorithm 4.2). The heuristic is similar to the one described in [27]. The basic idea behind the heuristic is that, in the inner iteration, at least  $\delta$  buyers get assigned to an object while  $\epsilon_{\text{local}}$  converges faster to a large value (line 6). In the outer loop (line 5),  $\epsilon_{\text{local}}$  will be reset again to a small value if the threshold has been exceeded. Thus, the approximate variant forces the auction algorithm to match a buyer faster than in the former  $\epsilon$ -scaling strategy. This aggressive  $\epsilon$ -scaling strategy is embedded in the main routines of PAA (see Algorithm 4.1). Algorithm 4.2, called  $\epsilon$ -PAA, delivers a maximal weighted matching. The heuristic terminates when every buyer has been matched or the prices for the objects are too expensive so the bids for unassigned buyers are

negative.

The behavior of  $\varepsilon$  over the number of iterations using the different  $\varepsilon$ -scaling strategies, “adaptive,” “constant,” and “linear,” is illustrated in Fig. 4.4.

### 4.2.3 Optimality and Convergence

In general, the auction algorithm finds a maximal or maximum weighted matching depending on the  $\varepsilon$ -scaling strategy and the scaled edge weights in the bipartite graph.

Using a constant  $\varepsilon > 0$ , the running time of the sequential auction algorithm (see Algorithm 3.1) is  $\mathcal{O}\left(n\Delta(\mathcal{G}_b)\frac{C}{\varepsilon}\right)$ , where  $n$  represents the number of objects,  $\Delta(\mathcal{G}_b)$  is the overall maximum number of edges incident to a buyer, and  $\frac{C}{\varepsilon}$  is the maximum number of auction iterations per object. When using a linear  $\varepsilon$ -scaling strategy and having integer-valued benefits the worst case complexity is

$$\mathcal{O}(nm \log(nC)), \quad (4.3)$$

where  $C = \max_{ij} |w_{ij}|$  in order to find a maximum weighted matching [29]. Regarding the quality of the matching, the sequential auction algorithm converges to an optimal assignment, if benefits  $w_{ij}$  are all integers,  $\varepsilon < \frac{1}{n}$ , and if the  $\varepsilon$ -complementary slackness condition

$$w_{ij_i} - p_{j_i} \geq \max_j \{w_{ij} - p_j\} - \varepsilon \quad (4.4)$$

holds for all assigned pairs  $(i, j_i)$  at termination of the algorithm [28].

It may happen that there is no perfect matching in the graph. Then, at some point, the maximum valuable profits for the free buyers become negative and the bid is invalid, as the price of the object is larger than its benefit. Then, the current matching is the final output of the auction algorithm.

By having integer-valued benefits, the parallel distributed auction algorithm (see Algorithm 4.1) satisfies the  $\varepsilon$ -complementary slackness condition for an asynchronous parallel auction algorithm on shared memory architectures due to the fact that in both variants bids are computed by out-of-date prices and price updates are disordered but synchronized at certain time steps [28].





## Chapter 5

---

# Software Implementation Aspects of PAUL

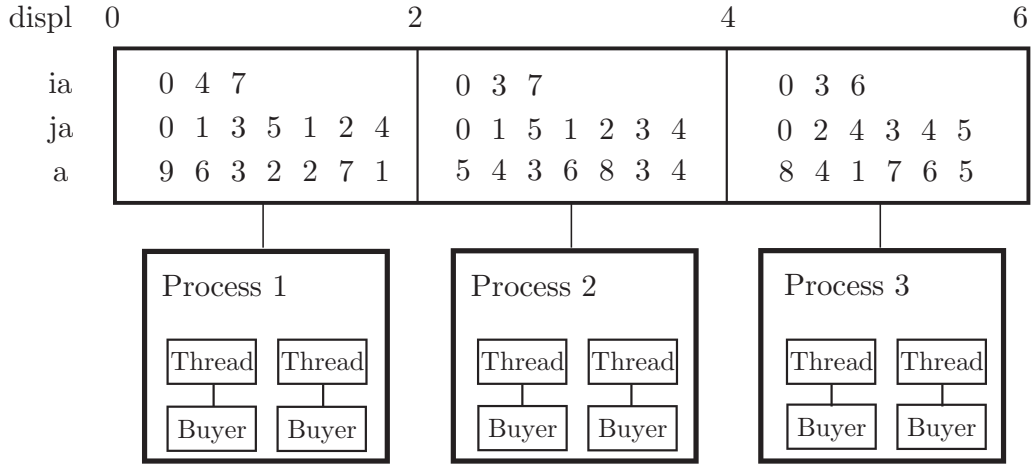
---

The parallel auction algorithm PAUL flows naturally into the hybrid MPI–OpenMP programming model. Hence, buyers and the connected objects are distributed among the available MPI processes and the bidding phase is accelerated using OpenMP threading.

### 5.1 Input Data

In PAUL, a bipartite graph is represented as a matrix stored in a 0-based CSR storage format as the complete source code of PAUL is implemented in C. Thus, the user must provide the application data stored in the three arrays “ia,” “ja,” and “a” (see Sec. 2.1). If the matrix  $\mathcal{A} = (a_{ij})$  is already distributed among the available processes, each MPI process must keep a row-wise rectangular part of the entire matrix where every distributed part should be again stored 0-based. In order to construct the entire matrix from the distributed parts, the user must offer a distribution vector “displ” which contains the starting indices of rows with respect to the entire matrix. In Fig. 5.1, an example is given for a distributed CSR matrix when 3 MPI processes are available. On each process, free buyers are assigned to the available OpenMP threads in order to compute the bids concurrently.

The matching obtained is coded in a permutation vector where position  $i$  and the entry at the position  $i$  of the permutation vector build a



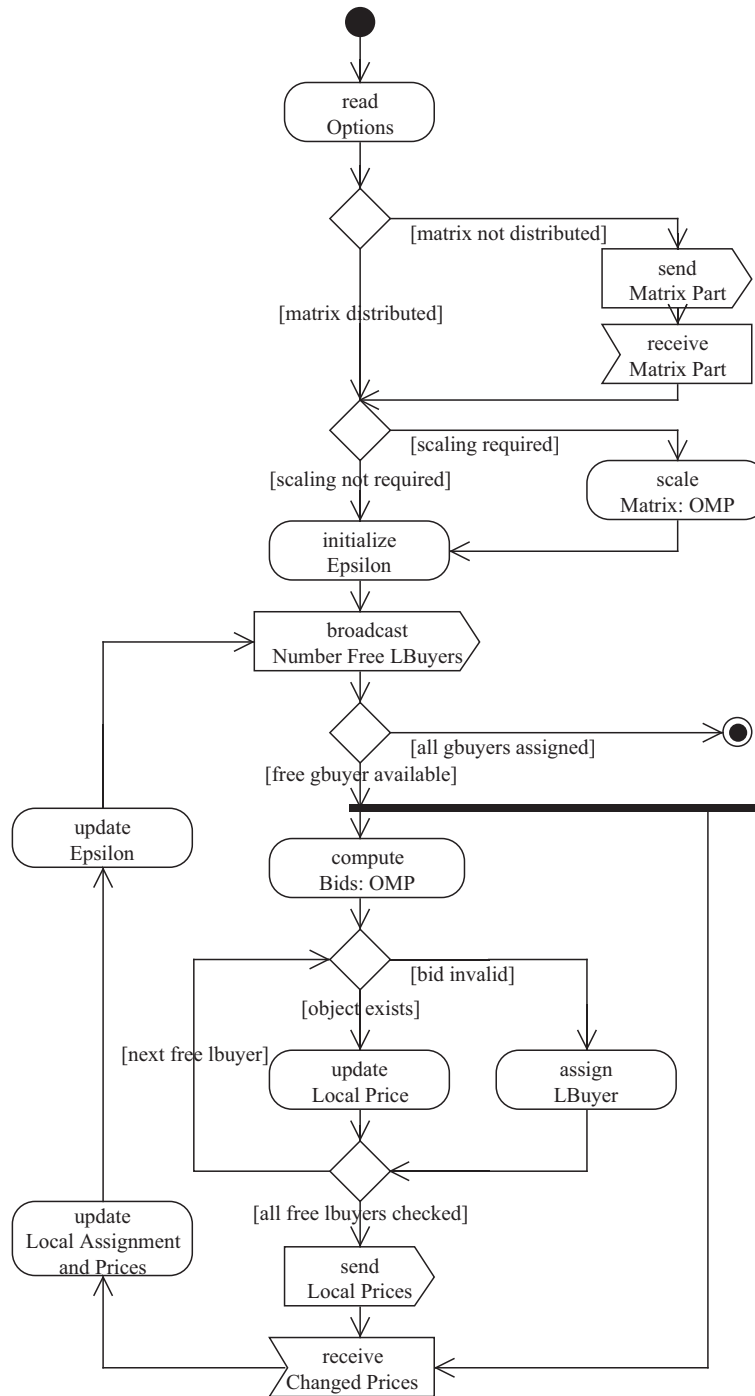
**Figure 5.1:** Mapping of a Distributed CSR Matrix to Processes and Threads.

matching pair. Additionally, scaling vectors  $r$  and  $p$  are returned. If a perfect weighted matching has been obtained, the scaling vectors scale the matrix so that all entries are between  $-1$  and  $1$  with  $a_{ij} \leftarrow a_{ij} \cdot r_i \cdot p_j$ .

## 5.2 Work Flow in PAUL

Before calling the parallel software library the user has to make sure that MPI is initialized and finalized in the calling program. In the call of the library the user must provide the matrix stored in the CSR storage format, allocate memory for permutation and scaling vectors, optionally give a distribution vector, and an option array which enables, for instance, different  $\varepsilon$ -scaling mechanisms (see Appendix A.2).

The work flow of PAUL from the view of all involved MPI processes is visualized in Fig. 5.2. At the beginning, the option array is read by all MPI processes (see state “read Options”). If a value in the array is negative, a default value is set for this particular option. In the options it can be determined, for instance, if the matrix is already distributed among the available MPI processes and if scaling of the matrix is required as the matrix contains values which are smaller than 0. Note that the product of the weights of the matched edges is maximized when the internal scaling mechanism is applied to the matrix. The scaling mechanism can be accelerated when using OpenMP threading to find the maximum entry per row and to perform the actual scaling of the values (see state “scale Matrix: OMP”). Afterwards, every MPI process sets up the parameter for the



**Figure 5.2:** Activity diagram of PAUL using the hybrid MPI-OpenMP programming model.

$\varepsilon$ -scaling mechanism which depends on the chosen strategy as described in Sec. 4.2.

A buyer who is only visible to a single MPI process is called “lbuyer.” The term “gbuyer” abbreviates “global buyer.” Every MPI process is responsible for a set of lbuyers and competes with the other processes for the assignment of all lbuyers to valuable objects. If there is no free lbuyer available, the process only updates the local prices and acts if a local buyer gets unassigned again. If at least one local buyer is unmatched on an MPI process (“broadcast Number Free Lbuyers”), the highest and second-highest bids are computed (see compute Bids: OMP). In order to speed up the computation of the bids, all available OpenMP threads are spanned over the free buyers. However, if the highest bid is negative (“bid invalid”), the lbuyer will remain unassigned in the subsequent auction iterations and can be removed from the list of unassigned buyers. If valuable objects exist for free buyers, the bids are computed and the prices are locally updated (“update Local Price”). Then, the prices need to be exchanged to determine the global winner for the object and, finally,  $\varepsilon$  is updated in an auction iteration. As long as there exists an unmatched buyer (“free gbuyer available”), all MPI processes are involved in the algorithm.

## **Part III**

# **A Dense Subgraph Problem as Building Block in Numerical Linear Algebra**



## Chapter 6

---

# Design of Scalable Hybrid Linear Solvers

---

Solving  $\mathcal{A}\mathbf{x} = \mathbf{b}$ , where  $\mathcal{A} \in \mathbb{R}^{n \times n}$  being sparse and nonsingular,  $\mathbf{b} \in \mathbb{R}^n$  a given right-hand side, and  $\mathbf{x} \in \mathbb{R}^n$  an unknown vector, is one of the re-occurring and time-consuming kernels in computational science. Direct solvers and iterative solvers combined with strong preconditioners are considered to be the two major research routes to solve ill-conditioned, large-scale, and sparse linear equation systems efficiently. These linear equation systems typically arise in optimization and simulation processes, in particular, after discretization of partial differential equations (PDEs), and need to be solved efficiently. As direct solvers lack on performance scalability and iterative solvers on robustness when solving large-scale 3-D problems [37], *hybrid solvers* have emerged, which are combining the advantages of direct and iterative solvers in new types of parallel linear solvers. It is shown that graph algorithms are essential in the construction of a strong preconditioner for the hybrid liner solver PSPIKE.

### 6.1 Hybrid Linear Solvers

Parallel direct solvers are known to be numerically reliable for general linear equation systems, but the performance and memory scalability is quite limited in particular for large 3-D PDEs [94]. If the problem requires a solution with machine precision accuracy, several parallel dense linear algebra (BLAS-3) based implementations are available as software,

such as MUMPS [6], PARDISO [210], SUPERLU [150], and WSMP [108]. On the other hand, if an approximative solution is sufficient, preconditioned iterative solvers [198] are the method of choice. Iterative methods and, in particular, Krylov-subspace methods like BICGSTAB [228] or GMRES [199], progressively enhance the quality of the solution, rely on sparse linear algebra kernels like sparse matrix–vector multiplication, tend to be easier to parallelize than direct methods, and considerably reduce the memory requirement compared to direct methods. However, the convergence rate of iterative methods is dependent on the properties of the linear system. Therefore, searching for a powerful preconditioner [25, 208] is an on-going endeavor to meliorate the convergence properties for certain problem types as each problem structure favors its unique preconditioner.

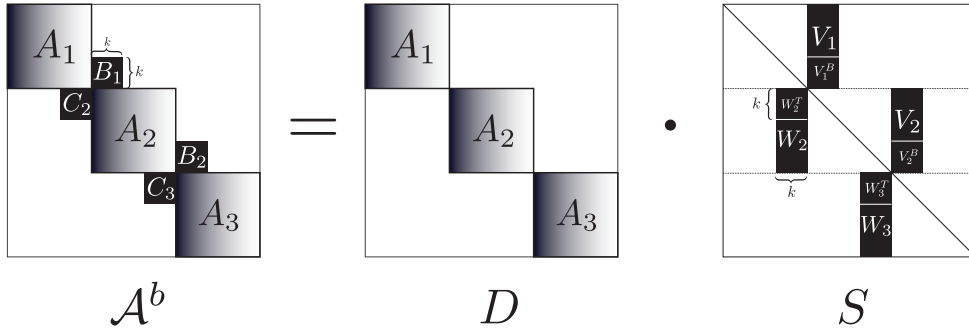
Parallel hybrid linear solvers — that take advantage of the strengths of both direct and iterative methods — offer approaches to overcome these dilemmas. The design of such solvers is generally governed by a canonical domain decomposition technique, i.e., by partitioning the matrix into block matrices with or without overlap. Then, the key idea is to apply a direct linear solver to individual blocks independently, and control the overall convergence with an outer preconditioned iterative solving scheme. In the inner direct solving scheme, the linear equation systems including the so-called *interior* entries of a block matrix, are solved independently via a parallel direct solver. In the outer preconditioned iterative solver scheme the solution of the system with the remaining so-called *interface* entries is approximated using a *Schur complement* approach. Recently, a few parallel hybrid solvers have been proposed which differ in the reordering of the matrix based on solutions of efficient graph algorithms, and in the approximation of the Schur complements [89, 99, 162, 194, 238].

Among these approaches the hybrid sparse linear solver PSPIKE is one of the most auspicious approaches to solve large-scale linear equation systems on massively parallel architectures. PSPIKE will be discussed in detail in the Secs. 6.2 and 6.3.

## 6.2 PSPIKE — A Scalable Hybrid Linear Solver

The hybrid solver PSPIKE combines a preconditioned iterative solver, here BICGSTAB, with a parallel direct solver, here the software library





**Figure 6.1:** SPIKE factorization of a sparse banded matrix  $\mathcal{A}^b$  in a block diagonal matrix  $D$  and a spike matrix  $S$ .

PARDISO. In the preconditioning step of the iterative solver, the key ingredient is a new kind of factorization scheme where PARDISO plays a central role.

### 6.2.1 The SPIKE Algorithm

In contrast to common direct factorization schemes like  $LU$ - or Cholesky factorization, in PSPIKE the SPIKE factorization  $\mathcal{A}^b = DS$  is employed, where  $\mathcal{A}^b$  is a banded matrix extracted from  $\mathcal{A}$  after applying several reordering strategies,  $D$  is a sparse block diagonal matrix, and  $S$  is the dense “spike” matrix [26, 51, 69, 148, 162, 170, 175, 191, 192, 200, 201]. This factorization is schematically illustrated using three processes in Fig. 6.1.  $\mathcal{A}^b$  is split into block diagonal matrices  $A_1, A_2, A_3$  and coupling block matrices  $B_1, B_2, C_2, C_3$ . Each process hosts a diagonal block and performs the respective factorization of a block using a sparse direct linear solver. The size of each coupling block is determined by the bandwidth  $k$  of the matrix  $\mathcal{A}^b$ . Thus, the sparse coupling block matrices  $B, C$  are of size  $k \times k$  and comprise the nonzero elements which are not already covered by square block diagonal matrices  $A_1, A_2, A_3$ . The spike matrix  $S$  contains dense spike matrices  $V_1, W_2, V_2, W_3$ , which have a width of  $k$ . Each process owns the spikes  $V, W$  corresponding to coupling block matrices  $B, C$ . The bottom ( $\mathcal{B}$ ) and top ( $\mathcal{T}$ )  $k \times k$  tips  $V_1^{\mathcal{B}}, V_2^{\mathcal{B}}$  and  $W_2^{\mathcal{T}}, W_3^{\mathcal{T}}$ , respectively, are highlighted due to their crucial role in PSPIKE. The diagonal elements of  $S$  are 1.

In SPIKE, the block diagonal matrix  $D$  and the spike matrix  $S$  are the components to obtain a solution of the linear equation system  $\mathcal{A}^b \mathbf{x} = \mathbf{b}$ .

In the forward diagonal solve, the linear system

$$D\mathbf{g} = \mathbf{b} \quad (6.1)$$

is solved with a parallel direct solver for all the diagonal blocks in parallel, as each diagonal block is independent of each other. If a diagonal block is singular (or close to being singular), diagonal boosting is applied to the linear system. It is assumed that each diagonal block  $A_i$  of  $\mathcal{A}^b$  is using an  $LU$ -factorization.

In the backward spike solve, solution vector  $\mathbf{g}$  is the new right-hand side in the linear equation system

$$S\mathbf{x} = \mathbf{g}, \quad (6.2)$$

in which  $S$  is the spike matrix, and  $\mathbf{x}$  is the solution of the system  $\mathcal{A}^b \mathbf{x} = \mathbf{b}$ . The spike matrix  $S$  is computed from  $S = D^{-1} \mathcal{A}^b$  when solving the linear systems

$$L_i U_i [V_i, W_i] = \left[ \begin{pmatrix} 0 \\ B_i \end{pmatrix}, \begin{pmatrix} C_i \\ 0 \end{pmatrix} \right] \quad (6.3)$$

in parallel. The first process computes only  $V_1$  and the last process  $p_K$  computes only  $W_K$ . The direct solver PARDISO solves the systems with the  $2k$  right-hand sides comprising the coupling block matrices  $B$  and  $C$  without explicitly storing the zeros.

Unfortunately, the spikes  $V$  and  $W$  are dense, and solving the linear systems (6.3) with dense linear solvers deteriorates the performance of the hybrid linear solver. Additionally, the memory requirement is increased as  $n_i k$  elements have to be stored in each spike. In PSPIKE a crafty new approach is being applied where only bottom and top tips  $V^B, W^T$  of spikes  $V, W$  are computed in the reduced spike system

$$L_i U_i [V_i^B, W_i^T] = \left[ \begin{pmatrix} 0 \\ B_i \end{pmatrix}, \begin{pmatrix} C_i \\ 0 \end{pmatrix} \right]. \quad (6.4)$$

PARDISO computes these tips by reordering the linear system and taking advantage of zero blocks in the coupling block matrices. The right-hand side and the tips of the spikes are stored in a compressed form. The bottom and top solutions  $\mathbf{x}^B, \mathbf{x}^T$  of  $\mathcal{A}^b$  can be calculated by solving the following system of size  $2k(p_K - 1)$ :

$$\begin{pmatrix} I & V_i^B & 0 & 0 \\ W_{i+1}^T & I & 0 & V_{i+1}^T \\ W_{i+1}^B & 0 & I & V_{i+1}^B \\ 0 & 0 & W_{i+2}^T & I \end{pmatrix} \begin{pmatrix} \mathbf{x}_i^B \\ \mathbf{x}_{i+1}^T \\ \mathbf{x}_{i+1}^B \\ \mathbf{x}_{i+2}^T \end{pmatrix} = \begin{pmatrix} \mathbf{g}_i^B \\ \mathbf{g}_{i+1}^T \\ \mathbf{g}_{i+1}^B \\ \mathbf{g}_{i+2}^T \end{pmatrix}. \quad (6.5)$$

**Algorithm 6.1:** *Outline of SPIKE algorithm for Solving  $\mathcal{A}^b \mathbf{x} = \mathbf{b}$* 

**Input:** Diagonal block matrix  $D$ , spike matrix  $S$ , coupling blocks  $B, C$ , right-hand side  $\mathbf{b}$   
**Output:** Solution  $\mathbf{x}$

- 1: Solve via direct solver Eq. 6.1
- 2: Partition  $\mathbf{g}_i$  into  $(\mathbf{g}_i^T, \mathbf{g}_i^M, \mathbf{g}_i^B)$ , send  $\mathbf{g}_i^{\{T,B\}}$  to corresp. process
- 3: Solve directly or iteratively Eq. 6.5
- 4: Send solutions  $\mathbf{x}_i^B, \mathbf{x}_i^T$  to corresp. process
- 5: Solve via direct solver Eq. 6.6

The top and bottom solutions  $\mathbf{x}^B, \mathbf{x}^T$  are now subtracted from the right-hand side of the original system so that the systems (6.6) can be solved independently in parallel,

$$L_i U_i \mathbf{x}_i = \mathbf{b}_i - \begin{pmatrix} 0 \\ I^B \end{pmatrix} B_i \mathbf{x}_{i+1}^T - \begin{pmatrix} I^T \\ 0 \end{pmatrix} C_i \mathbf{x}_{i-1}^B, \quad (6.6)$$

where the first process sets  $\mathbf{x}_0^B = 0$  and the last process sets  $\mathbf{x}_{K+1}^T = 0$ . In Algorithm 6.1, the five steps of the SPIKE algorithm are outlined. It solves linear equation systems composed of diagonal block and coupling block matrices in machine precision where at least two direct solver calls (line 1, line 5), and two communication phases are needed (line 2, line 4).

The system in Eq. 6.5 (line 3) can be either solved directly or iteratively, depending on the required solution accuracy and application [192]. Existing schemes are directed at how to solve this system. When the matrix is diagonally dominant, the “truncated” SPIKE algorithm solves only Eq. 6.7 instead of Eq. 6.5 which omits the coupling blocks  $V_{i+1}^T$  and  $W_{i+1}^B$ . In the “recursive” SPIKE algorithm, Eq. 6.5 is directly solved via a recursive scheme which essentially partitions this system until a small coupling block system is obtained. The third scheme “on-the-fly” describes iteratively solving the system without explicitly generating the matrix. These different schemes are also implemented in a software library [176].

In this implementation of PSPIKE, the latter scheme is preferred to the other schemes for solving Eq. 6.5 as the linear systems need to be efficiently solved twice per outer BICGSTAB iteration, and the scheme is not depending on specific properties of the matrix. Fortunately, a strong preconditioner is already known for the “inner” iterative solver. Thus, pre-

conditioned BICGSTAB is employed with the preconditioner  $\begin{pmatrix} I & V_i^B \\ W_{i+1}^T & I \end{pmatrix}$ . Subsequently, the independent reduced systems

$$\begin{pmatrix} I & V_i^B \\ W_{i+1}^T & I \end{pmatrix} \begin{pmatrix} \hat{\mathbf{x}}_i^B \\ \hat{\mathbf{x}}_{i+1}^T \end{pmatrix} = \begin{pmatrix} \mathbf{g}_i^B \\ \mathbf{g}_{i+1}^T \end{pmatrix} \quad (6.7)$$

are solved via a dense direct solver such as LAPACK [181]. The stopping criterion for this inner BICGSTAB is given by

$$\frac{\|S\hat{\mathbf{x}} - \mathbf{g}\|_2}{\|\mathbf{g}\|_2} < \epsilon_{\text{in}}, \quad (6.8)$$

where  $\epsilon_{\text{in}}$  is the inner tolerance. Then,  $\hat{\mathbf{x}}_{i+1}^T$  is sent back to the successor process to obtain  $\tilde{\mathbf{x}}$ . The solving of the linear equation system (see Eq. 6.8) is referred to as the *inner* solve.

### 6.2.2 The PSPIKE Algorithm and Implementation Issues

In PSPIKE, the SPIKE factorization is the building block in the preconditioned iterative solver. The PSPIKE algorithm, which is outlined in Algorithm 6.2, is partly based on the work described in [161] and encompasses three phases: *preprocessing* (lines 1–5), *numerical hybrid factorization* (lines 6 – 9), and *solving* the linear system (lines 10–16) [205, 206]. In the preprocessing phase, all data structures are allocated for storing the preconditioner  $\mathcal{A}^b$ , the partitioned matrix  $\mathcal{A}^r$ , the spikes, and the reordering vectors. In order to be memory efficient, the preconditioner  $\mathcal{A}^b$  and the partitioned matrix  $\mathcal{A}^r$  are not stored explicitly; instead both are stored as a triple matrix structure: block diagonal matrices, coupling block matrices, and the remaining entries are kept as CSR matrices on the corresponding process. The permutation matrices  $\Pi_r$  and  $\Pi_c$  are filled with reorderings obtained from graph problems like a weighted graph matching, a graph partitioning, and a weighted dense subgraph problem. The modeling and solving of these graph problems are discussed in the next sections. The permutation matrices transform  $\mathcal{A}^b$  into  $\mathcal{A}^r$  and vice versa. Note, that lines 1–3 are performed by one master process which owns the basic data structures. Only the initialization of PARDISO and the parallel reordering algorithms are performed by all processes. As soon as the matrix is distributed among the processes, every process is actively involved in the computation.

In the numerical hybrid factorization (lines 6–9), the block diagonal matrices are *LU*-factorized via PARDISO, and additionally the linear systems involving the bottom and top spikes are solved. As the inner solving

**Algorithm 6.2:** *Outline of Hybrid Solver PSPIKE for Solving  $\mathcal{A}\mathbf{x} = \mathbf{b}$* 

**Input:** CSR matrix  $\mathcal{A}^{n \times n}$ ,  $\mathbf{b}$ , bandwidth  $k$ , #RHS, tolerance  $\epsilon_{\text{out}}$   
**Output:** Solution  $\mathbf{x}$

- 1: Initialize solver  $\triangleright$  *init direct solver; I/O and error handling*
- 2: Read  $\mathcal{A}$  and initialize data structures
- 3: Create  $\mathcal{A}^r = K_f Q_m P_{s_1} D_r A D_c P_{s_2} K_f^T$  and generate reorderings  
 $\Pi_r = K_f Q_m P_{s_1}$  and  $\Pi_c = P_{s_2} K_f^T$
- 4: Split  $\mathcal{A}^r$  row-block-wise and distribute  $\mathcal{A}_i^r$  to process  $i$
- 5: Extract square diagonal blocks  $A_i$  and  $k \times k$  coupling blocks  $B_i, C_i$   
from  $\mathcal{A}_i^r$   $\triangleright$  *construct  $\mathcal{A}^b$*
- 6: *LU* factorization of block diagonal matrices  $A_i$  with direct solver
- 7: Solve directly (Eq. 6.4)
- 8: Send  $k \times k$  dense matrix  $W_i^T$  to predecessor process  $i - 1$
- 9: *LU*-factorization of inner preconditioner (Eq. 6.7)
- 10: Reorder and scale right-hand side with  $\mathbf{b}^r = \Pi_r D_r \mathbf{b}$  and distribute RHS to corresponding process
- 11: Preprocessing of parallel SpMV (see Algorithm 6.3, lines 2–4)
- 12: **while**  $\frac{\|\mathcal{A}^r \mathbf{x}^r - \mathbf{b}^r\|}{\|\mathbf{b}^r\|} \geq \epsilon_{\text{out}}$  **do**
- 13:     Outer iteratively solve via preconditioned BICGSTAB
- 14:      $\triangleright$  *including preconditioner call  $\mathcal{A}^b \tilde{\mathbf{x}} = \mathbf{z}$  (see Algorithm 6.1) and parallel SpMV (see Algorithm 6.3, lines 6–9)*
- 15: **end while**
- 16: Gather, scale back, and reorder  $\mathbf{x}^r$  to  $\mathbf{x} = D_c \Pi_c \mathbf{x}^r$

step needs the preconditioner (see Eq. 6.7), the top dense spikes of size  $k$  need to be communicated to the neighboring process. In order to build a scalable solver, it is recommended setting  $k \leq 1000$  as otherwise the communication of the dense blocks dominates the computational part. In the last step of the factorization (line 9), this preconditioner is *LU*-factorized via LAPACK [181].

In the solution phase (lines 10–16), the right-hand side is reordered and scaled in correspondence to  $\mathcal{A}^b$  and  $\mathcal{A}^r$ , respectively, and distributed among the processes. The outer preconditioned iterative linear solver BICGSTAB receives  $\mathcal{A}^r, \mathcal{A}^b$ , bandwidth  $k$ , and right-hand side  $\mathbf{b}^r$  as inputs. Besides the BLAS [33] routines, the preconditioner call including the inner solve and the parallel sparse matrix–vector multiplication (SpMV) [212] are compute intensive operations in BICGSTAB. Regarding the parallel SpMV, the distributed vector needs to be communicated

**Algorithm 6.3:** *Parallel SpMV*

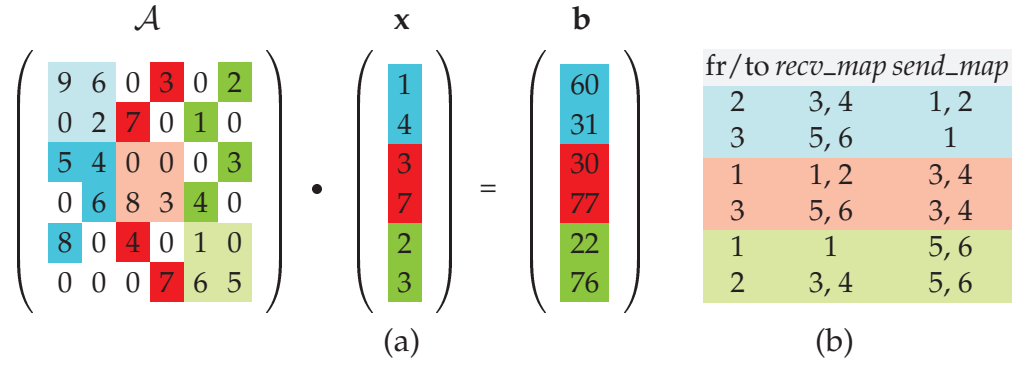
**Input:** Distributed CSR matrix  $\mathcal{A}^r$ , vector  $\mathbf{x}$   
**Output:** Solution  $\mathbf{z}$

- 1:  $\triangleright$  Preprocessing for SpMV
- 2: Determine indices of interface entries of local matrix  $\mathcal{A}_i^r$
- 3: Communicate indices to available processes
- 4: Store for each process: where and what to send in *send\_map* and from whom and from where to receive in *recv\_map*
- 5:  $\triangleright$  Actual SpMV
- 6: Send asynchronously interface entries by using *send\_map*
- 7: Perform local matrix–vector multiplication with interior entries and obtain local solution:  $\mathbf{z}_i \leftarrow \mathbf{z}_i + \mathcal{A}_i \mathbf{x}_i$
- 8: Receive interface entries by using *recv\_map*
- 9: Add matrix–vector multiplication with interface entries to  $\mathbf{z}_i$

to other processes in order to do the global matrix–vector multiplication as interface matrix entries are multiplied with nonlocal vector entries in every iteration. Obviously, the distributed vector could be globally synchronized among the processes in every iteration. However, this would lead to a tremendous communication overhead when only a few entries of the vector are required for the local matrix–vector multiplication.

In PSPIKE, the parallel SpMV is implemented in the two step procedure shown in Algorithm 6.3. Due to the fact that the structure of the matrix remains constant, a preprocessing step (lines 2–4) communicates the indices of interface entries to all processes and stores them in the local data structures *send\_map* and *recv\_map*. This step is executed only once. The actual matrix–vector multiplication step (lines 6–9) overlaps communication of the nonlocal vector entries with the local computation of the multithreaded matrix–vector multiplication. Finally, the interface entries are received and added to the local SpMV solution.

In Fig. 6.2, the preprocessing step of the SpMV with the sparse matrix  $\mathcal{A}$  and vector  $\mathbf{x}$  is illustrated. The matrix  $\mathcal{A}$  is initiated and partitioned as before (see Fig. 2.5); vector  $\mathbf{x}$  is distributed among processes so that each process holds only two entries of the full vector. In the matrix  $\mathcal{A}$ , the entries, which are highlighted in the light colors blue, red, and green, are part of the locally independent matrix–vector multiplication and do not require any communication. However, nonzero interface entries, which are highlighted in dark red, blue, and green, require the communication



**Figure 6.2:** (a) Illustration of a distributed SpMV when using three processes. (b) Communication layout for send and receive.

of nonlocal vector entries of  $\mathbf{x}$  in order to do the matrix–vector multiplication. One solution to the issue is to gather the full vector among all processes. However, it is definitely not an efficient solution for sparse matrices as nonzero column entries in a distributed matrix do not need all entries of the vector  $\mathbf{x}$  and, thus, communication can be drastically reduced. The table in Fig. 6.2(b) is precomputed once in order to send local vector entries only to processes which require the entries for the matrix–vector multiplication. The first two rows contain the information of the first process, the third and fourth rows the information of the second process, and the last two rows the information of the third process. The first column *fr/to* of the table contains the other available processes, the second column *recv\_map* the entries which are required from the process in column *fr/to*, and the column *send\_map* contains the entries which are needed by process *fr/to*. For instance, process 1 requires the entries 5, 6 from process 3, but process 3 needs only entry 1 to compute the matrix–vector product.

In each preconditioner call, the SPIKE algorithm is applied to the linear systems with modified right-hand sides. The stopping criterion for PSPIKE is

$$\frac{\|\mathcal{A}\mathbf{x} - \mathbf{b}\|_2}{\|\mathbf{b}\|_2} < \epsilon_{\text{out}}, \quad (6.9)$$

where  $\epsilon_{\text{out}}$  represents the tolerance level of the solver.

The PSPIKE implementation can also deal with multiple right-hand sides, which affects the solving phase of the hybrid solver. Then, all vector operations in BICGSTAB are extended to matrix operations, and all linear systems in the outer and inner iterative solver are enlarged to sys-



tems with multiple right-hand sides. The hybrid solver converges if the residuals for all right-hand sides are sufficiently small.

### 6.3 Graph Problems in PSPIKE

In the preprocessing phase of PSPIKE (see Algorithm 6.2, line 3), the reordering of  $\mathcal{A}$  is of paramount relevance for the scalability of PSPIKE as the outcome of the preconditioner call with  $\mathcal{A}^b$  determines its convergence speed. Thus, reorderings must permute heavy-weighted entries into either the diagonal blocks  $A_i$  or the coupling blocks,  $B_i$  and  $C_i$ , respectively. Ideally, all weighted entries are contained within these blocks, in which case the hybrid Krylov-subspace solver converges in a few iterations. However, in practical applications this situation appears to be rather rare. Thus, a goal is to construct  $\mathcal{A}^b$  so that most of the heavy-weighted entries are enclosed in the block structures, whereas some light-weighted entries might not be.

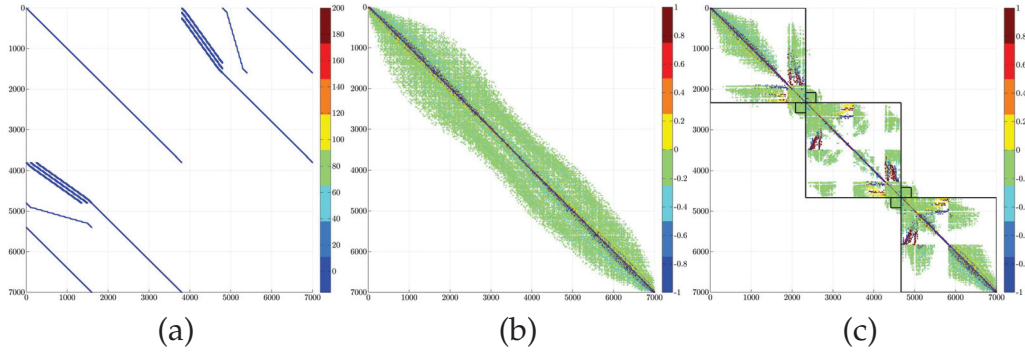
Combinatorial graph algorithms tackle the reordering problems posed in PSPIKE and can be formulated as well-known combinatorial optimization problems:

- guaranteeing (in practice) the nonsingularity of the block diagonal matrices can be modeled as a bipartite graph matching problem
- filling block diagonal matrices with heavy-weighted entries can be interpreted as a graph partitioning problem
- covering heavy-weighted entries with coupling blocks can be articulated as a weighted dense subgraph problem.

In Fig. 6.3(a), a typical structure of a saddle point matrix is visualized. The reorderings obtained from graph matching and spectral ordering results in a banded-like matrix (Fig. 6.3(b)). The load balancing issue is attained implicitly by the spectral ordering. The reordering of the densest subgraph problem completes the required PSPIKE structure with filled diagonal and coupling blocks (Fig. 6.3(c)).

The reordering of the first combinatorial problem, perfect weighted graph matching, permutes large entries onto the diagonal of the matrix via permutation matrix  $Q_m$ . Additionally, primal-dual matching algorithms [73, 109] provide scaling matrices  $D_r, D_c$  as a by-product generated from the dual variables, where entries on the diagonal are 1 or  $-1$ ,



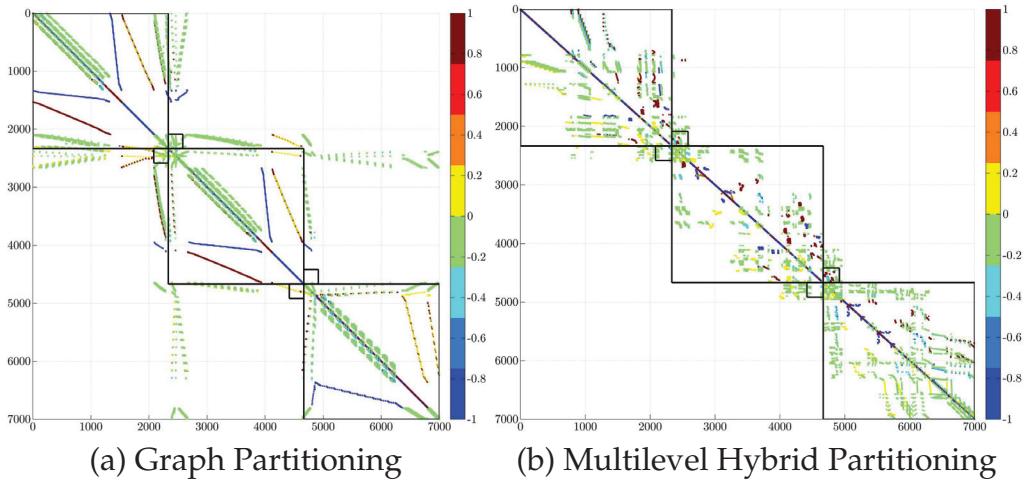


**Figure 6.3:** Saddle point matrix (a) and applied reorderings obtained through graph matching/spectral ordering (b), and solving dense subgraph problem (c).

and off-diagonal entries are between 1 and  $-1$ . The scaling causes the entire matrix to be close to diagonally dominant, and thus, the diagonal blocks, which are distributed among processes, are in most cases nonsingular [178].

The partitioner balances the load associated with the diagonal blocks while minimizing the number of heavy-weighted entries in off-diagonal blocks. Additionally, a specific requirement for a partitioner must be satisfied: interface entries in off-diagonal blocks should be reordered between neighboring diagonal blocks, the so-called *candidate blocks*  $B_i^c$  and  $C_{i+1}^c$  as heavy-weighted entries from the candidate blocks are considered for the reordering into the coupling blocks. However, there is no software available which can achieve such a reordering. It is required to combine the objective of load balancing for interior entries with the goal of bandwidth minimization for interface entries, but solely applying the permutation of a graph partitioner to the matrix will usually produce empty candidate and coupling blocks as shown in Fig. 6.4(a). In PSPIKE, two general options are implemented to deal with such a specific partitioning problem and to compute the permutation matrices  $P_{s_1}$  and  $P_{s_2}$ .

As a first option, a spectral heuristic implementation like MC73 [214] or TRACEMIN-FIEDLER [160] can be used as a partitioning routine as heavy-weighted entries are settled around the diagonal which implies that most of them reside on diagonal and candidate blocks, respectively. Afterwards, matrices are partitioned into the desired parts (e.g. row-wise), even though the number of available processes is not incorporated in the heuristic. Consequently, load balancing is only attained implicitly due to the fact that spectral heuristics reduce the bandwidth of a matrix and generate a symmetric reordering based on its Laplacian, i.e.,



**Figure 6.4:** Influence of Graph Partitioning in Reorderings for PSPIKE.

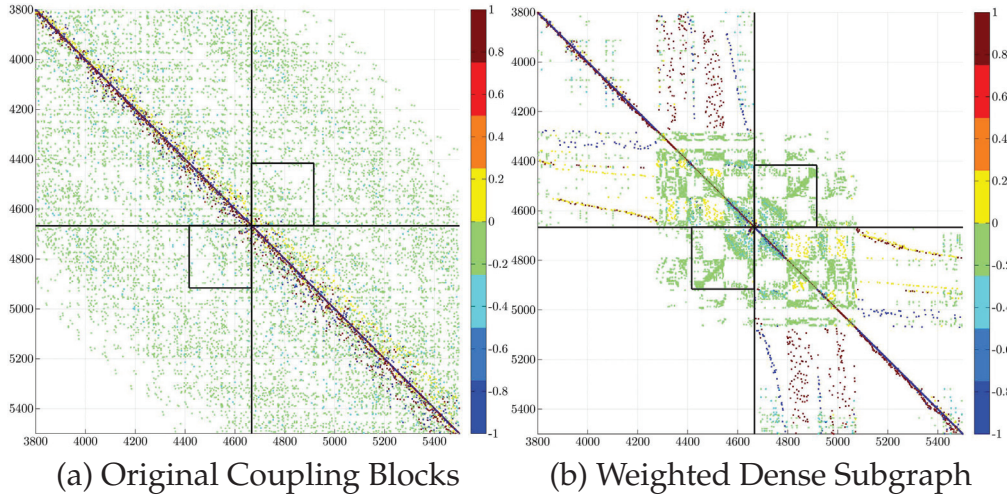
$$P_{s_2} = P_{s_1}^T.$$

The second option, *multilevel hybrid partitioning*, is to feed reorderings of a hypergraph partitioner (e.g., MONDRIAAN, PATOH), graph partitioner (e.g., METIS), or spectral orderings (e.g., TRACEMIN-FIEDLER) into a multilevel hybrid algorithm — the modified Sloan’s algorithm — that computes the profile of the matrix [144, 214]. Then, the matrix can be row-wise distributed. With this approach both goals, load balancing and bandwidth minimization, are taken into account. In Fig. 6.4(b), the obtained reordering of the saddle point matrix is wonderfully shown for three processes. Employing this approach in Algorithm 6.2, the reordering (in line 3) will change to

$$\mathcal{A}' = K_f R_o Q_m P_{s_1} D_r \mathcal{A} D_c P_{s_2} R_o^T K_f^T, \quad (6.10)$$

where  $K_f$  is the reordering obtained by an algorithm solving the weighted dense subgraph problem,  $R_o$  is the permutation matrix returned by the Sloan’s algorithm,  $Q_m$  is the reordering by the bipartite matching algorithm,  $P_{s_1}$  and  $P_{s_2}$  are the row and column permutation matrices returned by graph partitioner or spectral reorderings, and  $D_r$  and  $D_c$  are the scaling matrices obtained by solving the bipartite graph matching problem. Note that  $\Pi_r$  and  $\Pi_c$  are updated accordingly to the new ordering.

The third pillar of graph-based reordering techniques constitutes the weighted dense subgraph problem, which maximizes the number of the heavy weighted entries in coupling blocks  $B_i$  and  $C_{i+1}$  and fills the reordering  $K_f$ . Before performing the actual computation, the size of the



**Figure 6.5:** *Influence of Weighted Dense Subgraph in Reorderings for PSPIKE.*

candidate blocks,  $B_i^c$  and  $C_{i+1}^c$ , need to be determined. Each candidate block must be uncoupled from other candidate blocks in order to guarantee a feasible global reordering. In PSPIKE, the size of the matrix containing  $B_i^c$  and  $C_{i+1}^c$  is determined by the midpoint of  $A_i$  and by the midpoint of the next neighboring diagonal block  $A_{i+1}$ . Thus, candidate blocks are constructed in a nonoverlapping way.

When the reordering is applied to the matrix the numerical properties of the diagonal blocks should be untouched as PARDISO must be able to factorize the blocks. Thus, both candidate blocks  $B_i^c$  and  $C_{i+1}^c$ , and implicitly the coupling blocks,  $B_i$  and  $C_{i+1}$ , need to be optimized simultaneously but under the constraint that entries in diagonal blocks must remain in their blocks. More precisely, a symmetric permutation must be generated, with row permutations of  $B_i^c$  identical to column permutations of  $C_{i+1}^c$ , and column permutations of  $B_i^c$  identical to row permutations of  $C_{i+1}^c$ . This precondition is satisfied by creating a single rectangular candidate block  $BC_i^T = B_i^c + (C_{i+1}^c)^T$ . Then, the problem is to find a dense weighted submatrix of size  $k \times k$  in  $BC_i^T$ . This task can be mapped to the dense weighted subgraph problem when a single numerical value is assigned to each row and column of  $BC_i^T$ . In the next chapter, different strategies are discussed of how to discover a single value per row and column, and then how to solve this problem efficiently.

Recall that the overall goal of the reordering strategies is to permute as many heavy-weighted entries as possible into the PSPIKE structures, i.e., into the diagonal blocks  $A_i$  and the coupling blocks  $B_i, C_i$ . In order to

measure the quality of the reordering, every process  $i$  computes the *cover rate* ( $\text{co}$ ) of nonoverlapped weighted entries:

$$\text{co}_i = \frac{\sum_{(p,q) \in A_i^r} |a_{pq}| - \sum_{(p,q) \in A_i} |a_{pq}| - \sum_{(p,q) \in B_i, C_i} |a_{pq}|}{\sum_{(p,q) \in A_i^r} |a_{pq}|} \in [0, 1], \quad (6.11)$$

where  $A_i^r$  is the distributed row-block of matrix  $\mathcal{A}$ . In general, the closer  $\text{co}_i$  gets to zero on all processes, the faster — in terms of BICGSTAB iterations and, thus, computational time — PSPIKE converges, as the preconditioner then unfolds its full potential and controls the convergence process of the iterative solver. In Fig. 6.5(a), a zoom into the original coupling blocks  $B_2$  and  $C_3$  of Fig. 6.3(b) is presented. In Fig. 6.5(b), the reordering of the entries of the candidate blocks into the coupling blocks is illustrated. The blocks are densely filled with heavy-weighted entries. The entries in the diagonal block matrices  $D_2, D_3$  are still residing in their blocks, but the structural pattern of the blocks has changed with the reordering.

However, it might be the case that the overlapping of the diagonal and coupling blocks is not good enough, i.e.,  $\text{co}_i$  is large. In PSPIKE, one opportunity to resolve this issue is to put a well-proven preconditioner  $\mathcal{Z}$ , in addition to  $\mathcal{A}$ , into the hybrid solver where the reorderings permit, to construct the PSPIKE structure  $\mathcal{Z}^*$  of  $\mathcal{Z}$  with a small  $\text{co}$  value. Then within the BICGSTAB solver, the diagonal and coupling blocks of  $\mathcal{Z}^*$  are fed into the SPIKE factorization scheme while the actual iterative solver, including the matrix–vector multiplier, is still operating on the permuted and scaled  $\mathcal{A}_i^r$ . However,  $\mathcal{A}^r$  is built through the reorderings and scalings computed on the basis of  $\mathcal{Z}^*$ .

## Chapter 7

---

### Dense Subgraph Problem

---

An important subproblem in the preprocessing phase of PSPIKE is the filling of the  $k \times k$  coupling block matrices with heavy-weighted entries: candidate blocks  $BC^T$  are cut from the entire matrix to identify these entries [206]. These candidate matrices are the input for different heuristics which optimize the weight of the  $k \times k$  submatrix. Finding a weighted  $k \times k$  submatrix in an adjacency matrix  $\mathcal{A} = (a_{ij})$  can be formulated by the following quadratic program

$$\begin{aligned} \max \quad & \sum_{i=1}^{n_1} \sum_{j=1}^{n_2} a_{ij} x_i y_j \\ \text{s.t.} \quad & \sum_{i=1}^{n_1} x_i \leq k, \\ & \sum_{j=1}^{n_2} y_j \leq k, \\ & x_i, y_j \in \{0, 1\} \quad i = 1, \dots, n_1; j = 1, \dots, n_2, \end{aligned}$$

where decision variables  $x_i$  and  $y_j$  enable row  $i$  and column  $j$ , respectively.

In order to classify the combinatorial optimization problem, the matrix representation is viewed as a bipartite graph  $\mathcal{G}_b = (\mathcal{V}_1, \mathcal{V}_2, \mathcal{E}, w)$ . Every row  $i$  of the matrix  $BC^T$  is modeled as a vertex in  $\mathcal{V}_1$ , every column  $j$  is represented as a vertex in  $\mathcal{V}_2$ , and the absolute value of entry  $|a_{ij}|$  becomes the weight of the edge  $w_{ij}$  between the vertices  $i$  and  $j$ . The task is to choose  $k$  vertices of  $\mathcal{V}_1$  and  $k$  vertices of  $\mathcal{V}_2$ , resulting in the two

sets  $V_1^k$  and  $V_2^k$  so that  $\sum_{i \in V_1^k, j \in V_2^k} w_{ij}$  is maximized. In this chapter the bipartite graph and corresponding matrix representations are used interchangeably: choosing  $k$  vertices of  $V_1$  corresponds to  $k$  rows of  $BC^T$  and  $k$  vertices of  $V_2$  corresponds to the  $k$  columns.

This combinatorial problem is solved by two construction heuristics and an evolutionary algorithm. The underlying idea in the heuristics is that a single meaningful value is assigned to each row and column which indicates the overall importance of the row and column in the solution. The heuristics find a subgraph with a high weight although there is no guarantee regarding optimality.

First, the different quality measures for the importance of a row and column are described, and then the heuristics are outlined.

## 7.1 Quality Measures

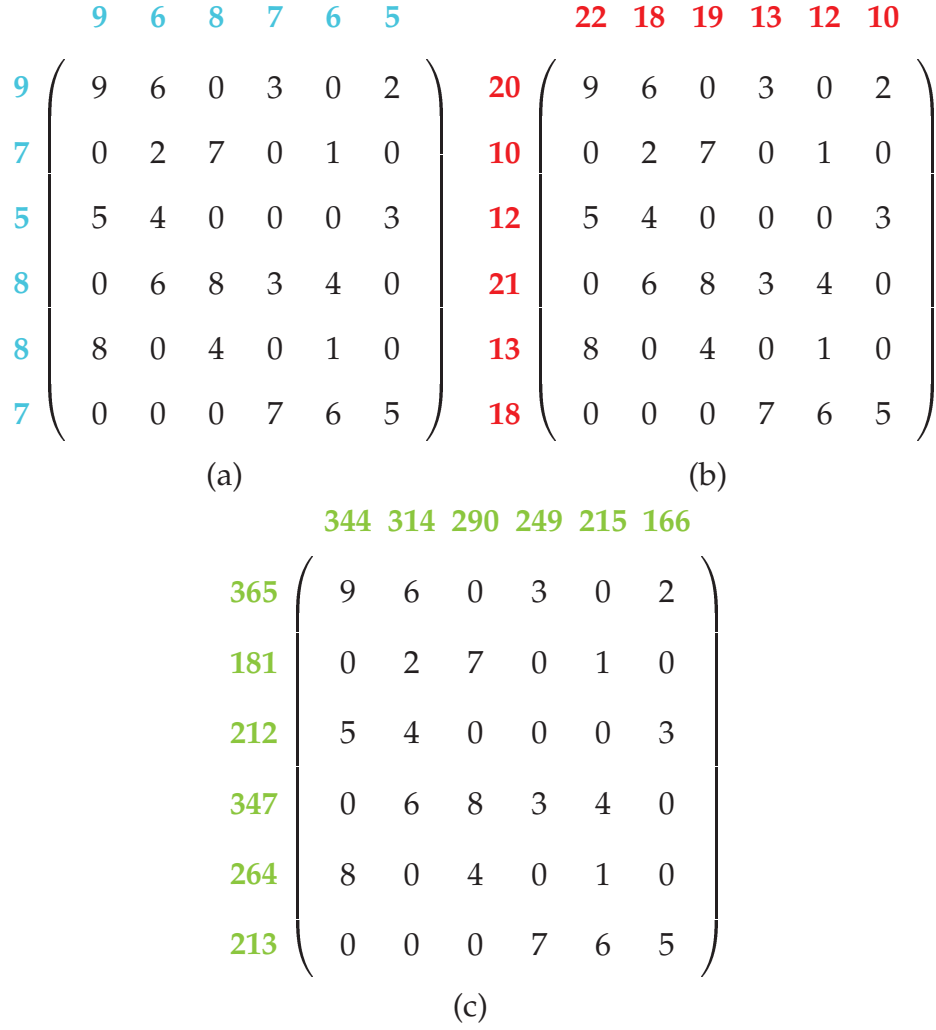
Three different mappings are evaluated to obtain a single numerical value for each row and column. The adjacency matrix  $BC^T = (a_{ij}) > 0$  can be described by the row set  $R = \{1, \dots, i, \dots, n\}$  and the column set  $C = \{1, \dots, j, \dots, m\}$ . A function  $\gamma : C \cup R \rightarrow \mathbb{R}^+$  is employed to assign to each row and column a single numerical value. In the following, the three different quality measures, “Max Entry,” “Sum,” and “Scalar Product,” which heavily influence the quality of the weighted subgraph heuristics are presented. As which quality measure is preferred to the others is dependent on the application, the quality measure must be seen in combination with the heuristic and data.

### Max Entry

In this strategy, the maximum entry per row and column, respectively, identifies its significance observing that the largest value may provide a good estimator. Hence,

$$\gamma_{\text{MaxEntry}}(s) = \begin{cases} \max_{j \in C} \{a_{sj}\}, & s \in R, \\ \max_{i \in R} \{a_{is}\}, & s \in C. \end{cases} \quad (7.1)$$

In Fig. 7.1, the maximum entries per row and column are shown in blue.



**Figure 7.1:** Visualization of the quality measures (a)  $\gamma_{\text{MaxEntry}}(s)$  (in blue), (b)  $\gamma_{\text{Sum}}(s)$  (in red), and (c)  $\gamma_{\text{Sp}}(s)$  (in green).

## Sum

This quality measure assigns the sum of its nonzero values for each row and column.

$$\gamma_{\text{Sum}}(s) = \begin{cases} \sum_{j \in C} a_{sj}, & s \in R, \\ \sum_{i \in R} a_{is}, & s \in C. \end{cases} \quad (7.2)$$

The sums over rows and columns are highlighted in red in Fig. 7.1. Alternatively, the quality measure  $\gamma_{\text{Sum}}(s)$  can be divided by the number of nonzero entries per row and column, respectively, in order to obtain a



mean value.

### Scalar Product

As a nonzero entry of a row or column will be only valuable if the corresponding column or row is picked as well, the scalar product between rows and columns might be a good quality indicator.

$$\gamma_{\text{Sp}}(s) = \begin{cases} \sum_{j \in C} a_{sj} \gamma_{\text{Sum}}(s), & s \in R, \\ \sum_{i \in R} a_{is} \gamma_{\text{Sum}}(s), & s \in C. \end{cases} \quad (7.3)$$

An application of this mapping is presented in green in Fig. 7.1. The mean of the scalar product can be also considered as a reasonable extension.

The time complexity to compute one of the quality measures is  $\mathcal{O}(m)$  due to the fact that all entries of the matrix are involved at least two times (per row and column value) in the calculation.

These are the basic evaluation mechanisms for the subsequent heuristics to decide if a row or column is selected or deleted.

## 7.2 Heuristics

In PSPIKE, currently three heuristics, two construction heuristics and an evolutionary algorithm, are implemented to solve the problem approximately. In the following, each of them will be discussed in detail.

### 7.2.1 FirstFit

Since including sorting in a greedy search method often retrieves a solution with an acceptable quality especially for combinatorial problems, it is also considered for the weighted submatrix problem. The heuristic FIRSTFIT is constructed as follows. First, one of the above sketched quality measures is applied to the matrix. Then, the rows and columns are sorted according to the numerical values independently of each other in decreasing order, and the  $k$  heaviest rows and columns are picked as the final index set for the coupling blocks. The major advantage of the greedy heuristic is its speed; the time complexity is dominated by the sorting procedure. However, the weight of the submatrix might be only suboptimal. For instance, for the matrix in Fig. 7.1, the greedy heuristic



**Algorithm 7.1:** *Heuristic DELETEmIN for the Weighted Submatrix Problem*

**Input:** Adjacency matrix  $BC^T$ , row and column set  $R, C$ , size  $k$   
**Output:** Row and column set  $R, C$

```

1: Sort  $R$  and  $C$  in ascending order
2:  $i \leftarrow 0; j \leftarrow 0$ 
3: while  $(|R| > k) \vee (|C| > k)$  do
4:   if  $(|R| = k) \vee ((|C| > k) \wedge (r_i > c_j))$  then
5:      $C \leftarrow C \setminus \{j\}$ 
6:     Update and maintain  $R, C$ 
7:      $j \leftarrow j + 1$ 
8:   else
9:      $R \leftarrow R \setminus \{i\}$ 
10:    Update and maintain  $C, R$ 
11:     $i \leftarrow i + 1$ 
12:   end if
13: end while

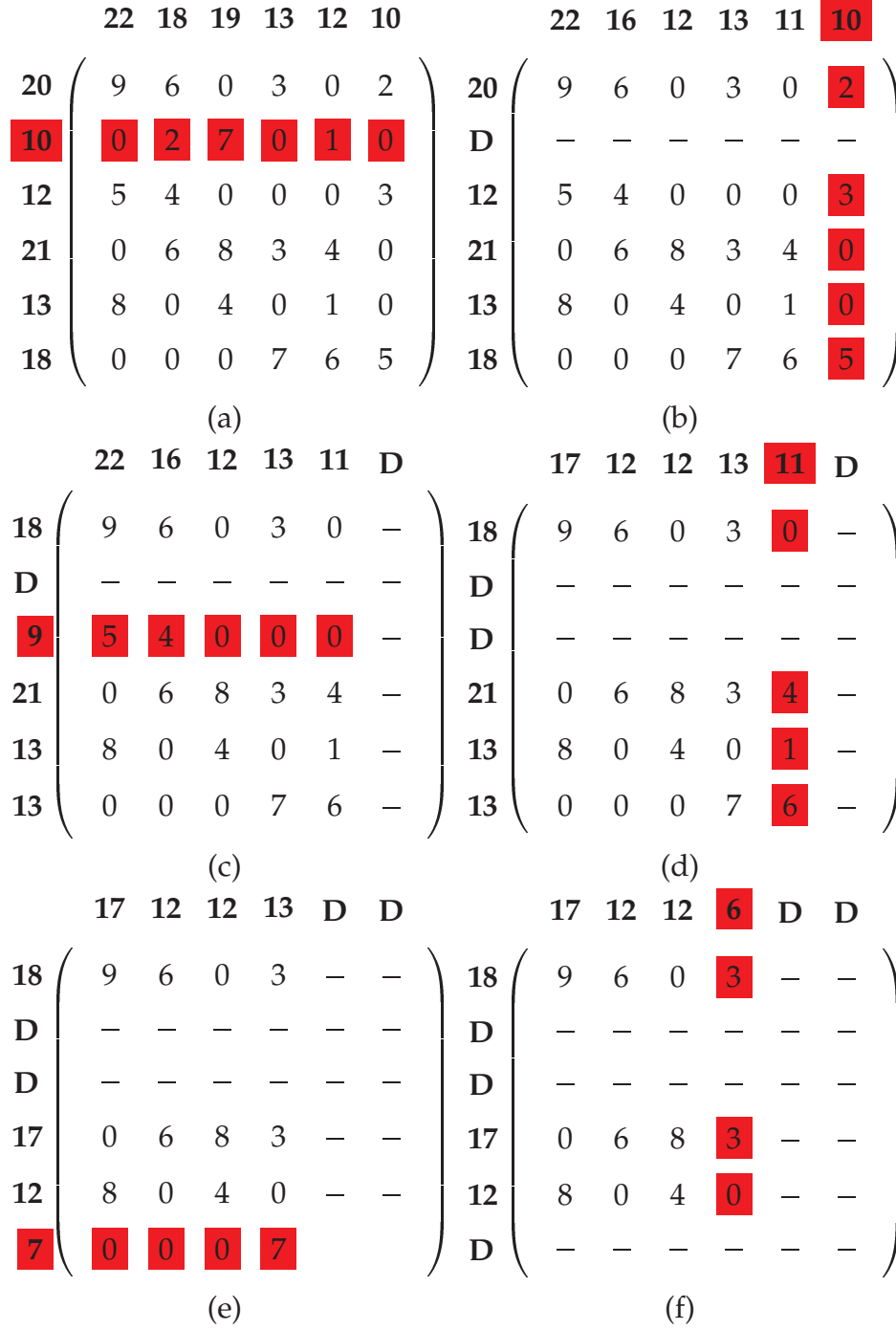
```

computes different solutions for each of the quality measures described above. Assume  $k = 3$ , the heuristic FIRSTFIT in combination with quality measure

- $\gamma_{\text{Sum}}(s)$  selects rows 1, 4, 6 and columns 1, 2, 3 with a total weight of 29
- $\gamma_{\text{MaxEntry}}(s)$  picks rows 1, 4, 5 and columns 1, 3, 4 with a total weight of 35
- $\gamma_{\text{Sp}}(s)$  chooses rows 1, 4, 5 and columns 1, 2, 3 with a total weight of 41 which is the optimal solution.

### 7.2.2 DeleteMin

A major disadvantage of the greedy heuristic is that  $k$  rows and columns are chosen concurrently, but the impact of the choice on other rows and columns is not incorporated in the heuristic. For instance, there might be a row which strongly connects to already chosen columns and would be an attractive candidate for the coupling blocks, but will not be considered as a candidate due to the greedy nature of FIRSTFIT. In the DELETEmIN heuristic this issue is elucidated by mimicking a construction heuristic.



**Figure 7.2:** Illustration of the DELETETEMIN heuristic for bipartite weighted dense  $k$ -subgraph problem with  $k = 3$ .

In Algorithm 7.1, the DELETETEMIN heuristic is outlined. To start, one of the quality measures is chosen to determine values of  $r_i, c_j$ . First, rows

$$\begin{array}{cc} \begin{pmatrix} 1 & 0 & 9 \\ 5 & 7 & 0 \\ 5 & 8 & 0 \end{pmatrix} & \begin{pmatrix} 10 & 0 & 1 \\ 0 & 10 & 9 \\ 0 & 0 & 10 \end{pmatrix} \\ \text{(a)} & \text{(b)} \end{array}$$

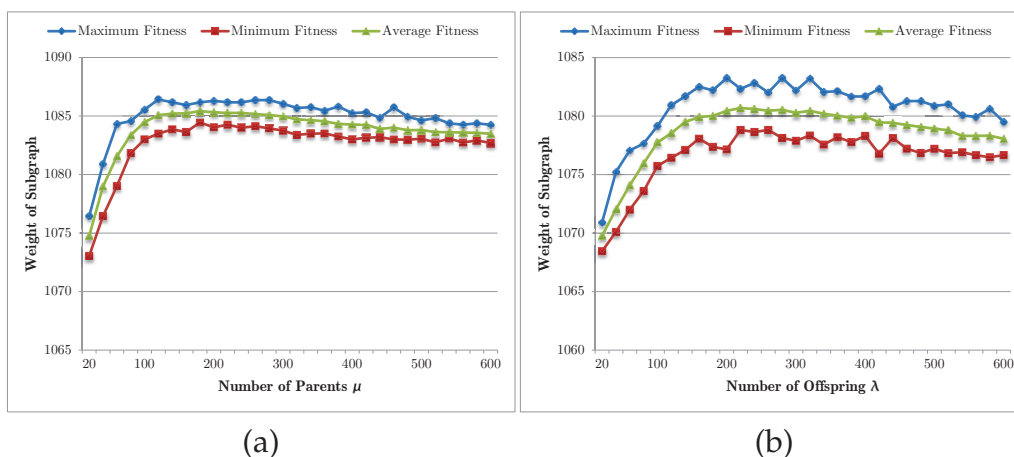
**Figure 7.3:** Illustration of two exemplary matrices.

$r_i \in R$  and columns  $c_j \in C$  are sorted in an ascending order with respect to the quality measure (line 1); thus, rows and columns with a lower contribution to the objective will be processed first. As long as the size of the submatrix is larger than  $k$ , the row or column with the currently least contribution is removed from the current row or column set (lines 4–12). Then, the single numerical value of “active” rows and columns is “outdated,” and row or column values depending on the removed row or column need to be updated at each iteration. Additionally, the sorting of the list needs to be adapted to the up-to-date scores. The time complexity of the procedure is  $\mathcal{O}(n^2 \log n)$  due to the initial sorting and maintaining of the sorted list in every iteration.

A step-by-step example for DELETETMIN is illustrated in Fig. 7.2. The numbers printed in bold in the first column and first row were computed by the quality measure  $\gamma_{\text{Sum}}(s)$ . In the heuristic, rows and columns with the currently smallest contribution are deleted successively (indicated in red), while remaining rows and columns are updated with the new values. For instance, in Fig. 7.2(a), the contribution of the second row is very low so that, in Fig. 7.2(b), the row is deleted, which is indicated with the “D”; columns 2, 3, and 5 were connected with the row 2 and the values need to be updated. After six iterations, rows 1, 4, 5 and columns 1, 2, 3 constitute the  $3 \times 3$  submatrix.

The results of the FIRSTFIT and DELETETMIN heuristics for this matrix are also visualized in Figs. 2.6(b) and (c), respectively. Although the DELETETMIN heuristic generates results with a better quality compared to the FIRSTFIT heuristic, it is not very efficient for large problems, even if  $k$  is rather small due to the time complexity.

The importance of the applied quality measure is demonstrated by two simple example matrices in Fig. 7.3. In Fig. 7.3(a), both heuristics DELETETMIN and FIRSTFIT will not find the optimal solution, which is 9, when using  $\gamma_{\text{MaxEntry}}(s)$  for  $k = 1$ . In Fig. 7.3(b), FIRSTFIT combined with  $\gamma_{\text{MaxEntry}}(s)$  computes the worst possible solution as it can choose the first column and the third row with a total weight of 0.



**Figure 7.4:** Relation from the number of drawings of offspring and parents to the fitness value.

### 7.2.3 Evolutionary Algorithms

As DELETMIN is computationally too expensive with the quadratic complexity for large problems, and the performance of FIRSTFIT is strongly influenced by the quality measures, and not robust enough, a global improvement heuristic would enhance the quality of the results. An appropriate choice for hard-to-solve combinatorial optimization problems is, for instance, an EA. Here, a  $(1 + 1)$ -EA is developed which generates a single offspring from a single parent by mutation and lets the individual with the better fitness survive [78]. Consequently, the fitness value of an intermediate solution never decreases and the convergence behavior of the algorithm heavily depends on the chosen mutation operator. In the  $(1 + 1)$ -EA, an individual is described by the row and column indices, which construct the  $k \times k$  submatrix. The fitness function evaluates an individual by the sum over the nonzero entries in the current  $k \times k$  submatrix.

### Representation and Initialization

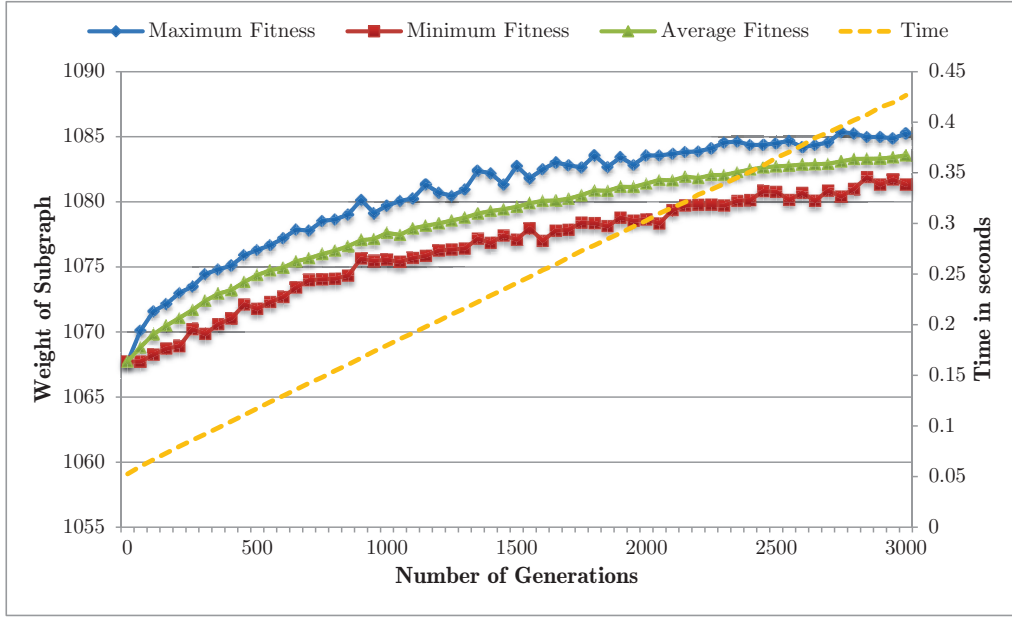
An important task is to determine an efficient genetic representation of the individuals in order to code meaningfully the solution of the weighted dense submatrix problem. Here, rows and columns, which were picked from the row and column sets, encode a solution. For instance, the string 125123 encodes that rows 1, 2, 5 and columns 1, 2, 3 were chosen for a submatrix of a size of  $k = 3$ . A general advantage of an EA lies in

the opportunity to feed initial solutions into the optimizer. Hence, the  $(1 + 1)$ -EA is supplied with the best candidates of multiple starting solutions, which are generated by the FIRSTFIT heuristic combined with the quality measures described above, and additionally with randomly generated individuals. In theory, the DELETMIN heuristic might also be picked as a candidate in the initialization phase, but in practice, the time complexity of this approach prevents its use for large matrices with a small  $k$ .

## Mutation and Selection

The mutation operator involves different quality measures in the generation process of a new individual. The underlying idea of the mutation operator is to exchange rows and columns with lower contributions with rows and columns with higher contributions, respectively. An offspring is chosen by randomly selecting  $\lambda$  rows and columns, and the indices of the “fittest” rows and columns are placed into the new offspring chromosome. From the parent individual,  $\mu$  row and column indices are randomly chosen, and the worst row and column indices are considered as potential replacement candidates. In Fig. 7.4, the typical influence on the fitness value of parameters  $\mu$  and  $\lambda$  is illustrated. In both experiments, Figs. 7.4(a) and (b), the number of generations is set to 5,000. The maximum, minimum, and average fitness values are evaluated over 50 runs for finding a weighted  $1,000 \times 1,000$  submatrix in a matrix of size 25,000 which has 8 entries per row on average. Two observations can be made from these experiments. First, the number of draws of parents and offsprings should be not too high as the exploration of promising rows and columns is too concentrated on a few candidates. Second, the number should be not too small as the process is biased with a high randomness rate. In this instance,  $\mu$  and  $\lambda$  attain a maximum fitness value with values of 180 and 120, respectively.

Assuming a parent and an offspring are chosen for evaluation, four possible outcome situations occur when comparing their fitness values: if the contribution of either a row or a column of the offspring is higher than a row or a column of the parent, then the corresponding row or column of the offspring replaces their counterparts in the parent. If the contributions to the fitness value of both, row and column, of the offspring dominate both contributions of the parent, the row and column in the gene of the parent are replaced by the offspring’s row and col-



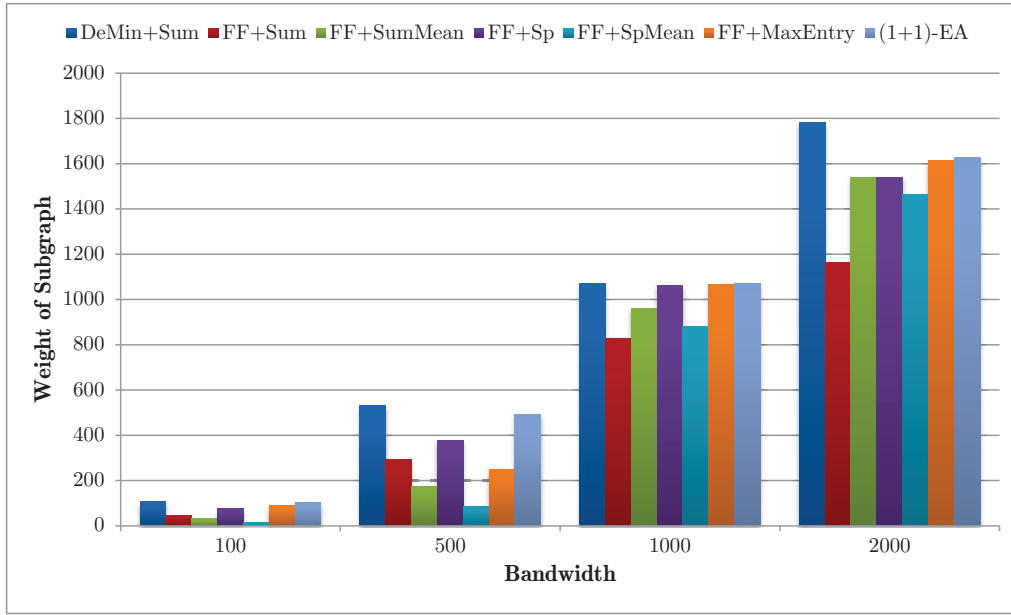
**Figure 7.5:** Convergence behavior of the  $(1 + 1)$ -EA with increasing number of generations.

umn. If the fitness value of the parent dominates the fitness value of the offspring, the parent will enter the next generation without any modification. In any case, the tournament starts again with the search for a new valuable offspring as long as the maximum number of generations has not been reached. As the fitness of an individual depends on the chosen quality measure, a switch between the quality measures enhances the exploration of the search space during the run of the EA.

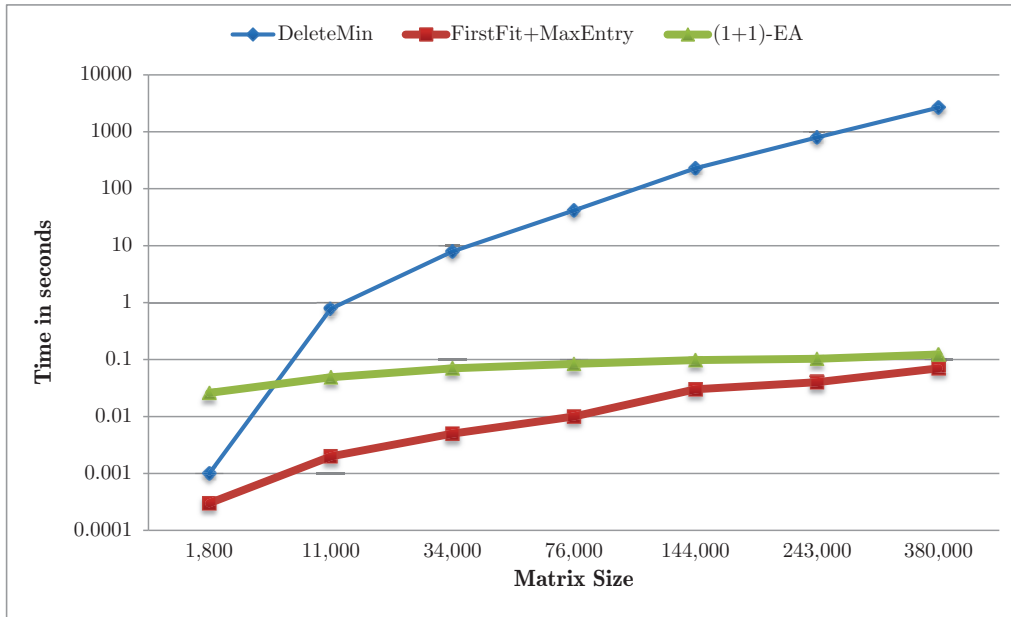
### Default Configuration and Convergence Behavior

As a configuration of the  $(1 + 1)$ -EA, the following values are proposed: the FIRSTFIT heuristic combined with the quality measures  $\gamma_{\text{MaxEntry}}$  and  $\gamma_{\text{Sp}}$  are used for the initialization of the individuals as they provide good starting solutions. The parameter  $\lambda$  is empirically set to  $k/5$ , while the number of offspring is fixed to  $\mu = 200$ . An individual is evaluated either by the  $\gamma_{\text{MaxEntry}}$  or by the  $\gamma_{\text{Sp}}$  fitness measure. In order to better explore the solution domain, the mutation operator switches between these two quality measures every 50 generations. The maximum number of generations is set to 500 by default.

In Fig. 7.5, a typical convergence behavior of the  $(1 + 1)$ -EA is pre-



(a)



(b)

**Figure 7.6:** Comparison of the quality and timings of the heuristics.

sented. The x-axis displays the number of generations, here shown up to 5,000 for illustration purposes. On the y-axes are the weights of the subgraph and the time to obtain the fitness value, respectively. It can be observed that the weight of the subgraph improves during the first 2,000

generations. Thus, the solution quality can be controlled by the number of generations, while the time depends only linearly on the number of generations.

### 7.3 Comparison of the Heuristics

The three heuristics, DELETMIN, FIRSTFIT, and the  $(1 + 1)$ -EA, need to be judged with respect to their quality and the time required for solving the weighted dense  $k$ -submatrix problem within PSPIKE. In Fig. 7.6, a typical behavior of the heuristics is presented. In Fig. 7.6(a) the heuristics are applied to a problem where the submatrix is of sizes  $k = 100, 500, 1,000$ , and  $2,000$ . The total weight of the  $k \times k$  submatrix is plotted on the y-axis. The heuristic DELETMIN always achieves a respectable result, but FIRSTFIT combined with quality measures  $\gamma_{\text{MaxEntry}}$  and  $\gamma_{\text{Sp}}$  retrieves solutions with similar qualities. Other quality measures in combination with FIRSTFIT turn out to be not as good. The solution quality of the  $(1 + 1)$ -EA is comparable to the DELETMIN heuristic, but when also considering the time required, the  $(1 + 1)$ -EA is faster than DELETMIN as shown in Fig. 7.6(b). The timings of FIRSTFIT using different quality measures are in the same range as the time using  $\gamma_{\text{MaxEntry}}$ . The solution timings of DELETMIN for relatively small matrix sizes are already inhibitive and would clearly dominate the solving time of PSPIKE. It can be concluded that the  $(1 + 1)$ -EA provides the best trade-off between quality and time for the considered heuristics, and is currently the best choice to solve the weighted subgraph problems within PSPIKE.



## Chapter 8

---

# Software Implementation Aspects of PSPIKE

---

PSPIKE is implemented for distributed memory architectures and follows the MPI–OpenMP programming paradigm. Thus, the source code must be compiled with an MPI compiler, linked with an OpenMP library, and has been tested with the PATHSCALE, GNU, and INTEL compiler suites using the optimization flag “-O3.” Calling PSPIKE requires that MPI is initialized and finalized in the calling program. Several external libraries for computing reorderings via graph algorithms and solutions of sparse and dense linear equation systems are implemented in PSPIKE. In this chapter, the interplay between these libraries is explained in detail.

### 8.1 Input Data

The square matrix of the linear equation system can be either symmetrically or unsymmetrically stored and is expected to be available in a 1-based CSR storage format as the computational kernel of PSPIKE is written in Fortran. The right-hand side of the linear system must be provided as a dense vector. The control over PSPIKE is given to the user via an external option file which enables, for instance, different reordering strategies and the debug output level of the library (see [Appendix A.1](#)). The bandwidth  $k$  and outer tolerance  $\epsilon_{\text{out}}$  can be put either via the interface or via the option file into the solver. The values of the bandwidth and tolerance are read from the option file if their values are set to 0 in

the interface.

## 8.2 The PSPIKE Phases

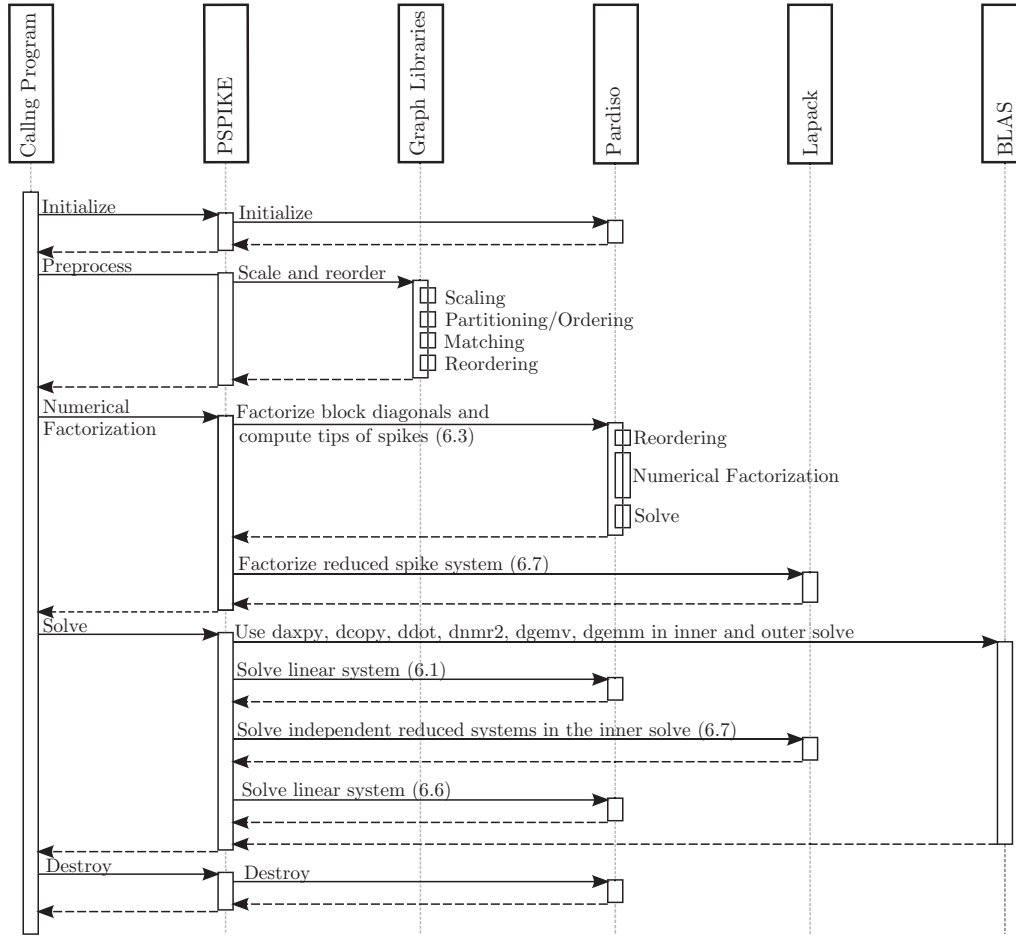
The implementation of PSPIKE consists of five phases: initialization, re-ordering, numerical hybrid factorization, solving, and destroy. The work flow of PSPIKE based on the five phases is illustrated in Fig. 8.1. The focal point of the figure is on the interplay between PSPIKE and the graph algorithm libraries LAPACK (e.g., [177, 181]), BLAS (e.g., [33, 177]), and PARDISO [209].

Starting from an external program, the initialization phase of PSPIKE must first be called. In this phase, the PSPIKE option file is read and internal methods and data structures are activated to realize the work flow determined by the enabled options. Additionally, the direct solver PARDISO is initialized. Typically, this phase is called only once for the entire solving process.

In the preprocessing phase, the main task is to reorder the matrix in order to obtain the PSPIKE structure with block diagonal and coupling block matrices. Therefore, several interfaces to existing implementations of graph algorithms for solving the graph matching, graph partitioning and ordering, and weighted subgraph problems are embedded into PSPIKE as described in Sec. 6.3. The software libraries provide permutation vectors which reorder nonzero elements into the PSPIKE structure. After each preprocessing step, auxiliary functions of PARDISO are used to apply the permutations to the matrix. The available combinations to stick reordering routines together will be discussed in the next section.

The numerical hybrid factorization phase involves all MPI processes in the computation and works as described in Sec. 6.2.2. It contains the factorization of the block diagonal matrices (6.1) and solving of sparse linear systems (6.3) via PARDISO in parallel. PARDISO internally reorders the matrix before factorize and solve the system. The solutions of the sparse linear system with the multiple right-hand sides (6.3) are the dense spikes which are factorized using the function `dgetrf` of LAPACK. However, this factorization is only required if the preconditioned inner solve is enabled in the options.

In the outer and inner solve, several routines of BLAS are used to compute vector and matrix operations in the iterative solver BICGSTAB. After each call of a BLAS routine the control flow is actually given to



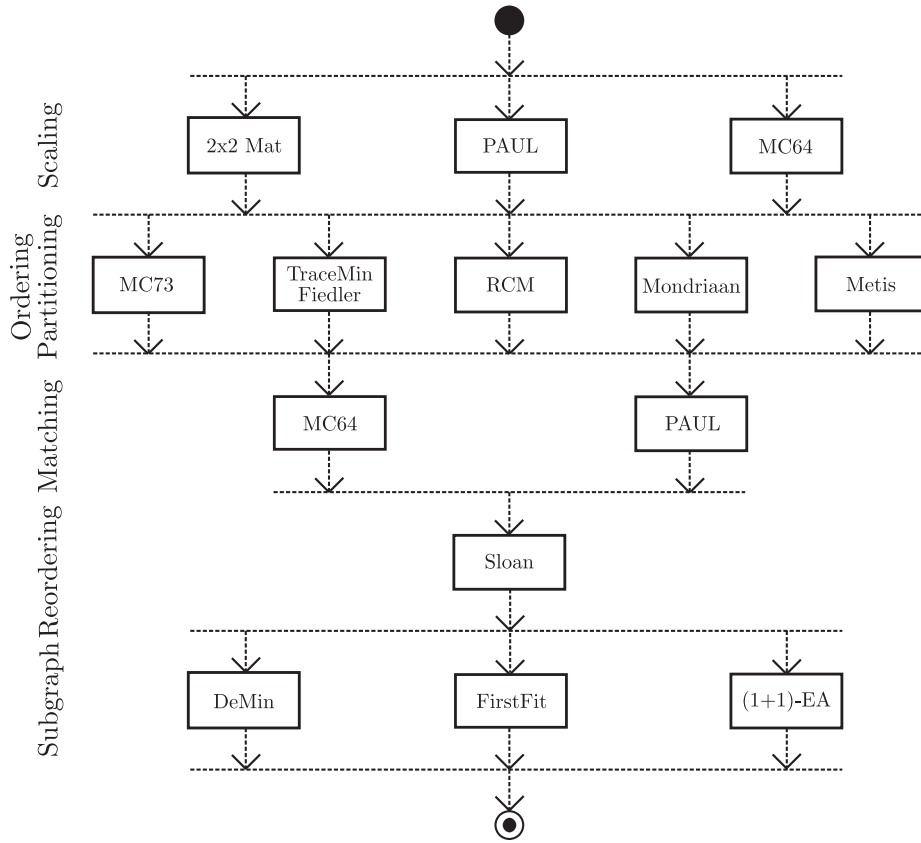
**Figure 8.1:** Sequence diagram showing the interplay between PSPIKE, external graph libraries, PARDISO, BLAS, and LAPACK.

PSPIKE which is not explicitly shown in this figure. In the outer solve, PARDISO is called twice in the preconditioner call (see Algorithm 6.1). In the inner solve, linear systems composed of the reduced spike system are solved using LAPACK (6.7).

In the destroy phase, the entire memory allocated by PSPIKE, LAPACK, and PARDISO is freed.

### 8.3 Combining Reordering Strategies

In the preprocessing phase, the reorderings applied to the matrix are crucial for obtaining a strong preconditioner. As the graph partitioning and spectral reordering heuristics provide solutions for  $\mathcal{NP}$ -hard problems,



**Figure 8.2:** Component diagram of the preprocessing phase of PSPIKE.

the quality of the solution considerably differs from application to application. Thus, it is recommended to find the best combination of reordering and scaling strategies in order to cover almost all entries of the matrix by the preconditioner. Note that PSPIKE automatically computes the quality of the preconditioner with regards to the coverage of the number of nonzeros and the number of heavy-weighted entries (see Eq. 6.11). In Fig. 8.2, the interfaces to follow scaling and reordering strategies are currently implemented which perform

- a scaling of the matrix entries ( $2 \times 2$  MAT [109], PAUL (see Sec. 4), MC64 [118])
- a spectral reordering (MC73 [118], TRACEMIN-FIEDLER [158], RCM [109]) or graph partitioning (MONDRIAAN [30], METIS [124]) of the corresponding graph to obtain heavy-weighted block diagonal matrices and nonempty candidate blocks

- a permuting of heavy-weighted entries on the diagonal via weighted graph matching algorithms (MC64 [118], PAUL (see Sec. 4))
- a reordering of the matrix to minimize its bandwidth and profile (SLOAN [118])
- a reordering of heavy-weighted entries into coupling blocks by solving the weighted dense subgraph problem (DEMIN, FIRSTFIT, (1+1)-EA (see Sec. 7)).

At each step, at most one of the available libraries can be optionally included into the entire scaling and reordering schemes. After each call of the external software libraries the matrix is immediately scaled using the dual variables of the matching algorithm, and also directly reordered using the permutation returned by the software libraries. It is possible through the PSPIKE option file to enable and disable scaling and reordering strategies (see Appendix A.1).



## **Part IV**

# **Data Intensive Applications**





## Chapter 9

---

# Applications

---

In this chapter, data intensive applications are described in graph similarity with data of protein-protein interaction networks and large web graphs, in arterial flow simulation, and in PDE-constrained optimization.

### 9.1 Graph Similarity

The detection of common patterns and properties in graphs, which is usually understood as *graph similarity*, is required by a broad range of applications such as chemoinformatics [195], pattern recognition [55], computer vision [152], bioinformatics [76], circuit design [197], and social network analysis [97]. Typically, internal data structures in these applications are mapped to graph-based representations; for instance, in bioinformatics or chemoinformatics where proteins or molecules are interpreted as vertices and their interactions are symbolized as edges.

There are different notions of graph similarity and, thus, different ways to compute similarity between graphs which are mainly based on graph isomorphism [186], graph edit distance [91], feature extraction [68], common subgraphs [82], and neighborhood similarity [35]. Other similarity measures can be found, for instance, in [183].

In *graph isomorphism*, two graphs are examined for having an identical structure, while *subgraph isomorphism* describes the search for a subgraph which is completely contained in another graph. A *maximum common subgraph isomorphism* attracts major attention in circumstances where the task is to find a single isomorphic subgraph to both graphs with

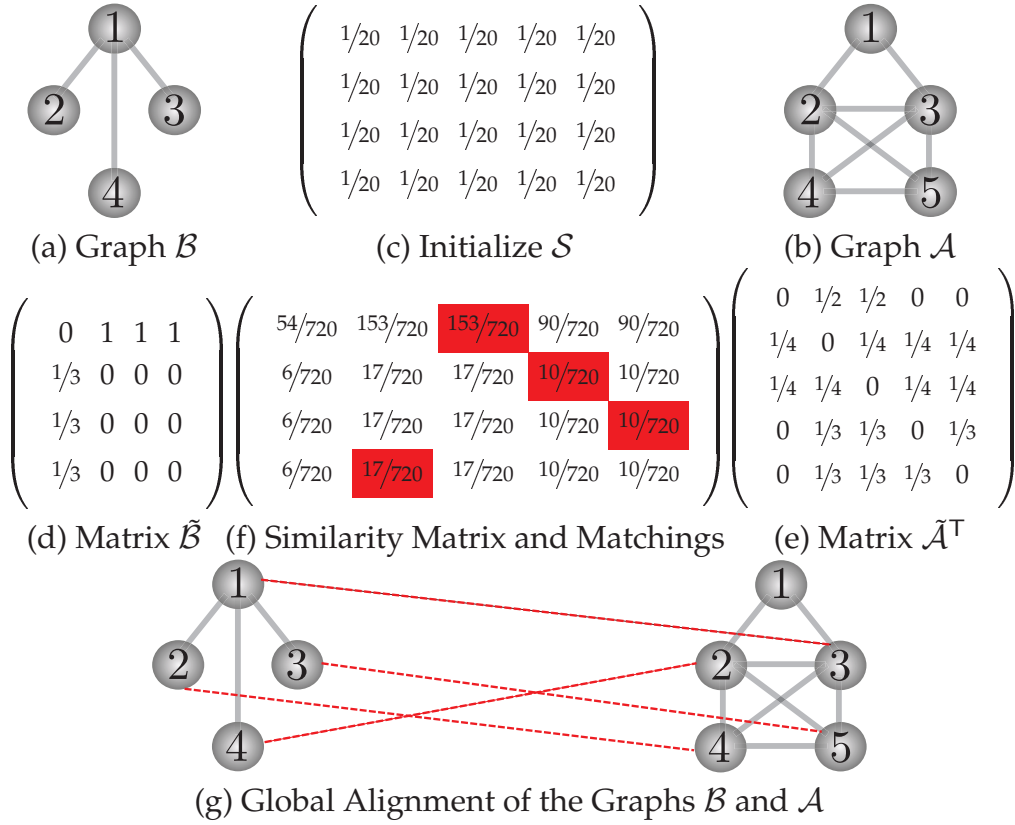
a maximum number of vertices, These problems are shown to be  $\mathcal{NP}$ -complete [22, 56].

In graph edit distance, the number of edit operations like insertion and deletion of nodes and edges, respectively, are counted in order to transform one graph into the other graph. A simple example is the hamming distance between two bit strings. This error-tolerant measurement is widely applied in pattern analysis and recognition. However, the computation of graph edit distance is  $\mathcal{NP}$ -hard, and the quality and time of existing heuristics are strongly dependent on the underlying cost function for the modification [241]. In feature extraction, statistical measurements like the degree distribution and betweenness centrality of vertices are major concerns which are often needed in social network analysis.

A large class of similarity computations rely on the general claim that the similarity of two vertices is determined by the similarity of their neighbors [35, 182]. In neighborhood similarity, vertices or edges are similar if the neighborhood of the vertices or edges are similar. The favorable principle is to compute a normalized similarity matrix using an iterative fixpoint method which considers the neighborhood of vertices. Then, vertex pairs are extracted using bipartite graph matching algorithms. A number of formulas and algorithms exist in the literature for both similarity computation and bipartite graph matching problems [239].

In the analysis of protein-protein interaction networks, it is of major interest to determine if two species have common functional and structural properties. Thus, protein structures of the species are *aligned* to each other and the *isomorphic* subgraph problem can be formulated in its graph representation. Hence, by finding similarities between proteins, newly sequenced genes can be inferred, new members of gene families can be predicted, and evolutionary relationships can be explored. Typically, large common subgraphs are preferred though small subgraphs also convey important insights into common biological functions of the two species.

General approaches to find such alignments can be separated into *local* and *global* alignment methods. In local alignment algorithms, unrelated subgraphs of two graphs are detected, where each subgraph is created independently of other subgraphs, i.e., a vertex can be assigned to different subgraphs. In contrast, global alignment algorithms are intended to find an overall alignment of the two networks without the inconsistencies of local alignment methods; i.e., each vertex should be matched with exactly one vertex. In this thesis, neighborhood similarity is considered



**Figure 9.1:** Illustration of graph similarity for  $\mathcal{S}^{(k)} \leftarrow \tilde{\mathcal{B}}\mathcal{S}^{(k-1)}\tilde{\mathcal{A}}^T$ .

in correlation with global alignment methods.

In the global alignment methods, a similarity matrix  $\mathcal{S} = (s_{ij})$  is constructed where  $s_{ij} \in [0, 1]$  denotes the similarity of vertex  $i$  of the first graph to vertex  $j$  of the second graph. The higher the value of  $s_{ij}$  the more similar are the two vertices. Then, the overall matching across the graphs is maximized by applying bipartite graph matching algorithms to  $\mathcal{S}$  where vertices of the first graph are mapped to similar vertices in the second graph.

Recently, a scalable algorithm called *network similarity decomposition* (NSD) has been designed that can be viewed as an accelerator for a large class of iterative similarity computation algorithms [134]. According to this algorithm, and given two adjacency matrices  $\mathcal{A} \in [0, 1]^{n_1 \times n_1}$ ,  $\mathcal{B} \in [0, 1]^{n_2 \times n_2}$ , the similarity matrix  $\mathcal{S} \in [0, 1]^{n_2 \times n_1}$  can be iteratively determined by using the power method approach which is similar to the actual PageRank calculation:

$$\mathcal{S}^{(k)} \leftarrow \alpha \tilde{\mathcal{B}}\mathcal{S}^{(k-1)}\tilde{\mathcal{A}}^T + (1 - \alpha)\mathcal{H}, \quad (9.1)$$

where  $\tilde{a}_{ij} = a_{ji} / \sum_{k=1}^{n_1} a_{jk}$ ,  $\tilde{\mathcal{B}}$  is defined analogously,  $\mathcal{H} \in [0, 1]^{n_2 \times n_1}$  is a preference matrix, and  $0 < \alpha < 1$  is a scaling factor where  $\alpha = 0$  implies no preference data are available, whereas, in contrast,  $\alpha = 1$  signals only preference data used. The update rule (see Eq. 9.1) can also be seen as an eigenvalue problem; when  $\mathcal{A}$  and  $\mathcal{B}$  are stochastic matrices with column sums of 1, the principal eigenvalue of Eq. 9.1 is 1, and converges in a finite number of iterations [219]. The preference matrix  $\mathcal{H}$  incorporates additional information; for instance, in protein-protein interaction networks  $\mathcal{H}$  can be filled with species specific values [34]. The scaling factor  $\alpha$  controls the influence of the preference matrix into the overall similarity matrix computation.

An illustration of the similarity computation is presented in Fig. 9.1 for two graphs  $\mathcal{A}$  and  $\mathcal{B}$  with the focus on the similarity computation  $\tilde{\mathcal{B}}\mathcal{S}\tilde{\mathcal{A}}^\top$ . The similarity matrix  $\mathcal{S}$  is uniformly initialized with  $\frac{1}{n_1 n_2}$  as no a priori information is known. Then, the adjacency matrices  $\tilde{\mathcal{B}}$  and  $\tilde{\mathcal{A}}$  are constructed according to the two input graphs  $\mathcal{A}$  and  $\mathcal{B}$  as follows: in both matrices, each row and column represents a vertex in its corresponding graph. The entry of a column for all adjacent vertices is set to the reciprocal of the number of adjacent vertices. For instance, the first column of  $\tilde{\mathcal{B}}$  represents vertex 1 in graph  $\mathcal{B}$  with the adjacent vertices 2, 3, 4; each entry is set to 1/3. The triple matrix product computation updates the similarity matrix  $\mathcal{S}$ , where each entry quantifies the similarity between two vertices of both graphs. Finally, weighted graph matching extracts best matching pairs which are highlighted in red. Note that there are several weighted matchings of equal quality in this example.

When substituting Eq. 9.1 into the subsequent iterations, the following equations are obtained after  $N$  iterations:

$$\begin{aligned}
\mathcal{S}^{(1)} &= \alpha \tilde{\mathcal{B}} \mathcal{S}^{(0)} \tilde{\mathcal{A}}^\top + (1 - \alpha) \mathcal{H} \\
\mathcal{S}^{(2)} &= \alpha \tilde{\mathcal{B}} \mathcal{S}^{(1)} \tilde{\mathcal{A}}^\top + (1 - \alpha) \mathcal{H} \\
&= \alpha \tilde{\mathcal{B}} \left( \alpha \tilde{\mathcal{B}} \mathcal{S}^{(0)} \tilde{\mathcal{A}}^\top + (1 - \alpha) \mathcal{H} \right) \tilde{\mathcal{A}}^\top + (1 - \alpha) \mathcal{H} \\
&= \alpha^2 \tilde{\mathcal{B}}^2 \mathcal{S}^{(0)} \left( \tilde{\mathcal{A}}^\top \right)^2 + (1 - \alpha) \alpha \tilde{\mathcal{B}} \mathcal{H} \tilde{\mathcal{A}}^\top + (1 - \alpha) \mathcal{H} \\
&\vdots \\
\mathcal{S}^{(N)} &= (1 - \alpha) \sum_{k=0}^{N-1} \alpha^k \tilde{\mathcal{B}}^k \mathcal{H} \left( \tilde{\mathcal{A}}^\top \right)^k + \alpha^N \tilde{\mathcal{B}}^N \mathcal{S}^{(0)} \left( \tilde{\mathcal{A}}^\top \right)^N.
\end{aligned} \tag{9.2}$$

The key factor in NSD is to initialize  $\mathcal{S}$  with a low-rank approximation of  $\mathcal{H}$ , i.e., to decompose  $\mathcal{H}$  into a sum of outer products of vector pairs, e.g., by computing a few singular vectors  $w_i, z_i$  ( $i = 1, \dots, s$ ) of its singular value decomposition, and iterating over the sum of scaled outer products of vectors. Assuming  $\mathcal{S}^{(0)} = \mathcal{H}$ , and  $\mathcal{H}$  can be decomposed into a given number  $s$  of outer products

$$\mathcal{H} = \sum_{i=0}^s w_i z_i^T \quad (9.3)$$

using, for example, Singular Value Decomposition. Furthermore, setting

$$w_i^{(k)} = \mathcal{B}^k w_i, \quad (9.4)$$

$$z_i^{(k)} = \left(\tilde{\mathcal{A}}^T\right)^k z_i, \quad (9.5)$$

and inserting  $w_i^{(k)}$  and  $z_i^{(k)}$  into Eq. 9.2, yields

$$\mathcal{S}_i^{(N)} = (1 - \alpha) \sum_{k=0}^{N-1} \alpha^k w_i^{(k)} z_i^{(k)T} + \alpha^N w_i^{(N)} z_i^{(N)T} \quad (9.6)$$

and, finally, the similarity matrix  $\mathcal{S}^{(N)}$  can be computed by

$$\mathcal{S}^{(N)} = \sum_{i=0}^s \mathcal{S}_i^{(N)}. \quad (9.7)$$

The entire computation of the similarity matrix using NSD is outlined in Algorithm 9.1. The advantage of computing the similarity matrix by NSD is that the algorithm is embarrassingly parallel as every process calculates blocks of the similarity matrix independently of other processes (lines 9–13). This computation clearly dominates the computation of the iterated singular vectors (lines 3–8). The overall time complexity of NSD is  $N \cdot s \cdot \mathcal{O}(n^2)$  where  $\mathcal{O}(n^2)$  is the time complexity to compute the matrix vector multiplication (see lines 6–7).

The second element in the similarity computation is bipartite graph matching which extracts similar pairs of the rectangular dense similarity matrix  $\mathcal{S}$ . Since similar pairs with a high similarity score are sought, weighted graph matching algorithms identify significant matchings  $m$ . The size of  $\mathcal{S}$  is determined by the size of  $\mathcal{A}$  and  $\mathcal{B}$  and, thus, especially in real life applications,  $\mathcal{S}$  can consist of millions of dense rows

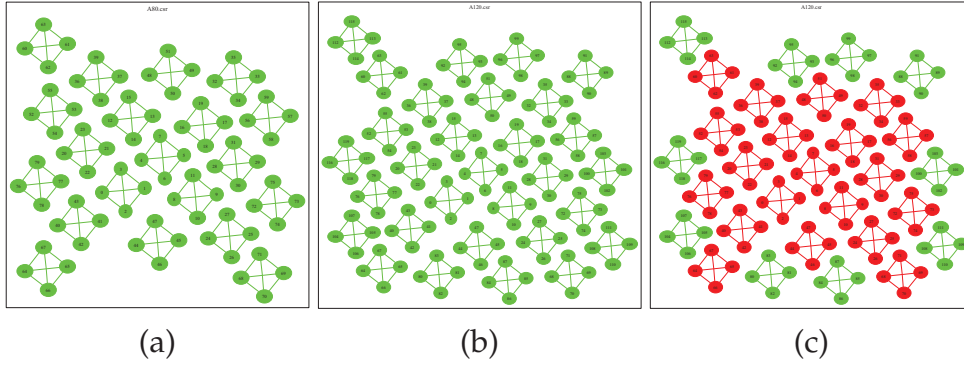
**Algorithm 9.1:** *Network Similarity Decomposition*

**Input:** Adjacency matrices  $\mathcal{A}, \mathcal{B}$ , vectors  $w_i, z_i$ , factor  $\alpha$ , #iterations  $N$   
**Output:** Similarity matrix  $\mathcal{S}^{(N)}$

- 1: compute  $\tilde{\mathcal{A}}, \tilde{\mathcal{B}}$
- 2: **for**  $i = 1$  to  $s$  **do**
- 3:    $w_i^{(0)} \leftarrow w_i$
- 4:    $z_i^{(0)} \leftarrow z_i$
- 5:   **for**  $k = 1$  to  $N$  **do**
- 6:      $w_i^{(k)} \leftarrow \tilde{\mathcal{B}} w_i^{(k-1)}$
- 7:      $z_i^{(k)} \leftarrow \tilde{\mathcal{A}} z_i^{(k-1)}$
- 8:   **end for**
- 9:    $\mathcal{S}_i^{(N)} \leftarrow 0$
- 10:   **for**  $k = 0$  to  $N - 1$  **do**
- 11:      $\mathcal{S}_i^{(N)} \leftarrow \mathcal{S}_i^{(N)} + \alpha^k w_i^{(k)} z_i^{(k)\top}$
- 12:   **end for**
- 13:    $\mathcal{S}_i^{(N)} \leftarrow (1 - \alpha) \mathcal{S}_i^{(N)} + \alpha^N w_i^{(N)} z_i^{(N)\top}$
- 14: **end for**
- 15:  $\mathcal{S}^{(N)} \leftarrow \sum_{i=1}^s \mathcal{S}_i^{(N)}$

and columns. To the best of our current knowledge there is no bipartite weighted graph matching implementation available able to deal with such huge dense matrices and compute high quality assignments in a few minutes. In this thesis, the parallel auction-based matching implementation PAUL is employed which computes weighted matchings efficiently.

Finally, common subgraphs can be detected by building the alignment graph from the matchings. In the alignment graph, each network is represented as a layer and additional edges are inserted into the graph if the structural properties of the subgraphs in two different networks are similar. In Fig. 9.2, two graphs — a graph with 80 vertices (a) and a graph with 120 vertices (b) — are automatically tested for subgraph isomorphism using NSD and bipartite graph matching. And indeed, the small graph is completely contained in the larger graph as highlighted in red Fig. 9.2(c). Assume there are given two graphs  $\mathcal{G}_1 = (\mathcal{V}_1, \mathcal{E}_1)$  and  $\mathcal{G}_2 = (\mathcal{V}_2, \mathcal{E}_2)$  with  $\mathcal{V}_1 = \left(v_p^1\right)_{p=1, \dots, n_1}$  and  $\mathcal{V}_2 = \left(v_q^2\right)_{q=1, \dots, n_2}$ . The quality of the extracted matchings  $m_i = (v_i^1, v_i^2)$ ,  $m_j = (v_j^1, v_j^2)$  is computed from

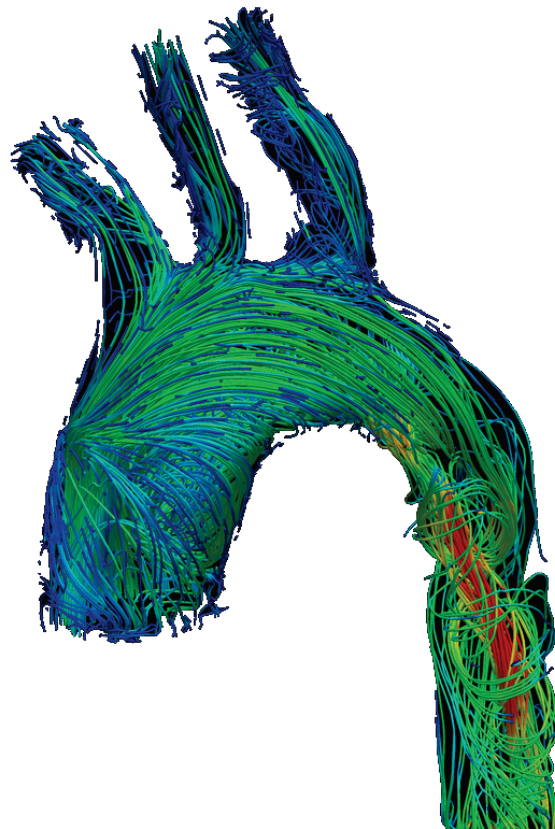


**Figure 9.2:** Subgraph isomorphism problem: alignment graph in red (c) of two input graphs with 80 vertices (a) and 120 vertices (b).

the alignment graph  $\mathcal{R} = (\mathcal{V}_{\mathcal{R}}, \mathcal{E}_{\mathcal{R}})$  with  $\mathcal{V}_{\mathcal{R}} = (m_k)$  and  $\mathcal{E}_{\mathcal{R}} \subseteq \mathcal{V}_{\mathcal{R}} \times \mathcal{V}_{\mathcal{R}}$ ; if  $m_i = (v_i^1, v_i^2)$  and  $m_j = (v_j^1, v_j^2)$  in  $\mathcal{G}_1 \times \mathcal{G}_2$  are two matchings, then  $(m_i, m_j) \in \mathcal{E}_{\mathcal{R}} \Leftrightarrow (v_i^1, v_j^1) \in \mathcal{E}_1$  and  $(v_i^2, v_j^2) \in \mathcal{E}_2$ .

When analyzing the alignment graph of two graphs, many metrics exist to evaluate the correspondence between two graphs. An overview over the existing functional measurements can be found in [149]. In general, the correspondences found by the weighted matchings can be evaluated from different perspectives. A topological evaluation of the matchings is the number of *conserved edges*, which corresponds to the number of edges in the alignment graph. Each conserved edge implies matching of an edge of one input graph to the edge of the other input graph. The existence of many conserved edges clearly increases the probability of them being part of extensive connected subgraphs; however, it might also be that they are parts of a larger number of connected subgraphs. An alternate measure called *similarity rate* is defined as the ratio of conserved edges over the minimum of the edges in the two graphs. Then, a similarity rate of 1 represents a full correspondence of the two graphs, and a number close to 0 represents a minor similarity between them. A different indicator for the similarity is the size of the common connected subgraphs in both graphs. There exists also a number of tools to derive application-dependent functional coherence between two graphs. For instance, for the evaluation of the alignment graph of biological networks, several ontologies exist like GENE ONTOLOGY [14], or PATH-BLAST [215] which standardize the descriptions and vocabulary of gene products in different databases. The advantages are that queries across





**Figure 9.3:** *The steady-state solution to the fluid flow through abdominal aorta; selected streamlines are colored by the magnitude of the velocity. Image courtesy of IT'IS Foundation, Zurich, Switzerland.*

different databases can be performed, and the resulting data are comparable to each other.

## 9.2 Arterial Flow Simulation

Aortic aneurysm is a serious disease which concerns the widening of the aorta to a value larger than  $1.5\times$  of the normal radius which leads to death if no appropriate medical treatment is begun.

In order to analyze the biochemical and biomechanical processes which are responsible for the formation of an aneurysm, pathology is concerned with the detection of the shear pattern and pressure distribution in the vulnerable location. However, conventional magnetic resonance imaging (MRI) is often not accurate enough; therefore, numerical simulations provide an attractive alternative for diagnosis and interventional plan-



ning. One of the most common types of aneurysm is abdominal aortic aneurysm, where flow conditions in conjunction with pressure and velocity distribution in the abdominal bifurcation are required to model the medical condition. The distributions in this domain can be found by numerically solving Navier-Stokes equations [185, 216]. In Fig. 9.3, the vortical flow of the velocity is illustrated.

Given viscosity  $\eta$  and density  $\rho$ , the Navier-Stokes equations for an incompressible fluid can be formulated as

$$\rho \left( \frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} \right) - \eta \nabla^2 \mathbf{u} + \nabla p = 0, \quad (9.8)$$

$$\nabla \cdot \mathbf{u} = 0, \quad (9.9)$$

where  $\mathbf{u} = \mathbf{u}(x, y, z)$  represents the velocity and  $p = p(x, y, z)$  is the pressure. A possible approach to solving these equations is provided in [40, 102]. There, a standard Galerkin finite element procedure is combined with a streamline diffusion to enhance the stabilization of the method. The time derivative is discretized via a backward Euler method, producing a nonlinear system of algebraic equations which need to be solved at every time step. The nonlinear systems are solved by Picard iteration resulting in a sparse linear system of equations of the form

$$\begin{pmatrix} A_u & B^T \\ B & 0 \end{pmatrix} \begin{pmatrix} \phi_u \\ \phi_p \end{pmatrix} = \begin{pmatrix} b_u \\ 0 \end{pmatrix}, \quad (9.10)$$

where  $\phi_u, \phi_p$  are velocity and pressure solution vectors, respectively, and  $b_u$  is the body force. The sparse unsymmetric matrix  $A_u$  results from the discretization of the time-dependent advection-diffusion equation and  $B$  is the discrete divergence operator.

Accelerating the entire simulation by solving the ill-conditioned, unsymmetric, sparse linear equation systems using a parallel linear solver, would be a remarkable gain in the medical treatment of the concerned patients. However, computer aided quantification is still not accepted for clinical decision making. The main issue preventing the method from being a clinical simulation tool is the prohibitive computational time, especially for solving the linear equation system (see Eq. 9.10).

### 9.3 Optimal Control of Partial Differential Equations

Optimal control aims at controlling a dynamical system while optimizing a certain objective. A dynamical system models commonplace physical phenomena in computational science like the distribution of heat or propagation of sound, waves, or fluid flow. These models are peculiarly described by PDEs. The optimization problem contains these PDEs as constraints, and optimizes *state* and *control* variables simultaneously. Since the objective function commonly includes a function to minimize the norm of the current state to a known goal state, e.g., a goal temperature for the state variable, a nonlinear and often nonconvex optimization problem occurs. The general form of a nonlinear optimization problem is

$$\min_{\mathbf{x}} f(\mathbf{x}) \quad (9.11a)$$

$$\text{s.t. } c_{\mathbb{E}}(\mathbf{x}) = 0, \quad (9.11b)$$

$$c_{\mathbb{I}}(\mathbf{x}) \geq 0, \quad (9.11c)$$

$$\mathbf{x} \geq 0, \quad (9.11d)$$

where the objective  $f : \mathbb{R}^n \rightarrow \mathbb{R}$ , the constraints  $c_{\mathbb{E}} : \mathbb{R}^n \rightarrow \mathbb{R}^p$  and  $c_{\mathbb{I}} : \mathbb{R}^n \rightarrow \mathbb{R}^q$  are considered to be twice continuously differentiable, and  $\mathbf{x}$  is the real-valued nonnegative decision variable. There are two strategies of how to embed the continuous PDE into the optimization problem.

The first strategy, *discretize-and-optimize*, discretizes the objective function and governing PDEs in space and time a priori and applies optimization algorithms to the large-scale problem. In the other strategy, *optimize-and-discretize*, first-order necessary and sufficient optimality conditions are created and the PDEs are then individually discretized. According to the *discretize-and-optimize* strategy, at first, the PDE is discretized (e.g., via finite differencing), and integrated into the optimization model as equality constraints (Eq. 9.11b). The discretization determines the size in terms of the number of decision variables of the particular optimization problem. In general, the finer the domain discretization the more accurate the solution for the PDE and, consequently, the global solution of the simulation and optimization process [115].

The inequality constraints (Eq. 9.11c) include restrictions on a set of control and/or state variables. As the size of the PDE-constrained opti-

mization problem can be increased to a problem with millions of decision variables, promising solution approaches are interior point methods which solve these large-scale nonlinear optimization problems to local optimality while ensuring global convergence.

In particular, a parallel interior point implementation, IPOPT [232], has been developed which is a state-of-the-art optimizer for large-scale nonlinear optimization problems, and forms the underlying nonlinear optimization software for the PDE-constrained optimization problems in this thesis. The developers of IPOPT have recently been awarded with the Wilkinson Prize: a prize to honor outstanding open-source software products every four years. In IPOPT, subsequent barrier problems are iteratively solved while using a filter mechanism to ensure optimality and feasibility of the solution. Integrating slack variables  $\mathbf{s}$  in the inequality constraint of optimization problem (Eq. 9.11c) and adding a logarithmic barrier term to the object function (Eq. 9.11a) results in the barrier subproblem

$$\min_{\mathbf{x}, \mathbf{s}} \quad f(\mathbf{x}) - \mu \sum_{i \in \mathbb{I}} \log s_i \quad (9.12a)$$

$$\text{s.t.} \quad c_{\mathbb{E}}(\mathbf{x}) = 0, \quad (9.12b)$$

$$c_{\mathbb{I}}(\mathbf{x}) - \mathbf{s} = 0, \quad (9.12c)$$

$$\mathbf{x} \geq 0, \quad (9.12d)$$

where  $\mu \rightarrow 0$  identifies a local optimal solution. The Lagrangian associated with this problem can be stated as

$$\mathcal{L}(\mathbf{x}, \mathbf{s}, \lambda) = f(\mathbf{x}) - \mu \sum_{i \in \mathbb{I}} \log s_i + \sum_{i \in \mathcal{E}} \lambda_i c_i(\mathbf{x}) + \sum_{i \in \mathbb{I}} \lambda_i (c_i(\mathbf{x}) - s_i), \quad (9.13)$$

and the first-order Karush-Kuhn-Tucker (KKT) optimality conditions are

$$\nabla_{\mathbf{x}} \mathcal{L} = \nabla_{\mathbf{x}} f(\mathbf{x}) + \sum_{i \in \mathcal{E}} \lambda_i \nabla_{\mathbf{x}} c_i(\mathbf{x}) + \sum_{i \in \mathbb{I}} \lambda_i \nabla_{\mathbf{x}} c_i(\mathbf{x}) = 0, \quad (9.14a)$$

$$\nabla_{\mathbf{s}} \mathcal{L} = -\mu S^{-1} \mathbf{e} - \lambda_{\mathbb{I}} = 0, \quad (9.14b)$$

$$\nabla_{\lambda_{\mathcal{E}}} \mathcal{L} = \mathbf{c}_{\mathcal{E}}(\mathbf{x}) = 0, \quad (9.14c)$$

$$\nabla_{\lambda_{\mathbb{I}}} \mathcal{L} = \mathbf{c}_{\mathbb{I}}(\mathbf{x}) - \mathbf{s} = 0, \quad (9.14d)$$

where  $S = \text{diag}(\mathbf{s})$  and  $\mathbf{e} = (1, \dots, 1)^{\top}$ .

IPOPT follows the “line search” optimization strategy, i.e., a new solution  $\mathbf{x}_{k+1}$  is computed by

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{d}_k, \quad (9.15)$$

where  $\mathbf{d}_k$  is the search direction and  $\alpha_k \in (0, 1]$  is the step size determined by the Armijo condition in iteration  $k$ . The search direction in (Eq. 9.15) is determined by  $\mathbf{d}_k = (\mathbf{d}_x, S\mathbf{d}_s)^\top$ , and the step size  $\alpha \in (0, 1]$  is found by a scaling mechanism in IPOPT.

Once a solution of the barrier subproblem (Eq. 9.12) satisfies the KKT conditions (Eq. 9.14), Newton's method is applied to Eq. 9.12 which involves solving the linear system

$$\left( \begin{array}{cc|cc} \nabla_{xx}^2 \mathcal{L} & 0 & \nabla_x \mathbf{c}_\mathcal{E}^\top & \nabla_x \mathbf{c}_\mathcal{I}^\top \\ 0 & \mu I & 0 & -S \\ \hline \nabla_x \mathbf{c}_\mathcal{E} & 0 & 0 & 0 \\ \nabla_x \mathbf{c}_\mathcal{I} & -S & 0 & 0 \end{array} \right) \begin{pmatrix} \mathbf{d}_x \\ \mathbf{d}_s \\ \mathbf{d}_{\lambda_\mathcal{E}} \\ \mathbf{d}_{\lambda_\mathcal{I}} \end{pmatrix} = - \begin{pmatrix} \nabla_x \mathcal{L} \\ -\mu \mathbf{e} - S\lambda_\mathcal{I} \\ \mathbf{c}_\mathcal{E} \\ \mathbf{c}_\mathcal{I} - \mathbf{s} \end{pmatrix}. \quad (9.16)$$

The Newton system — also referred to as KKT or saddlepoint system — is symmetric, indefinite, and ill-conditioned.

Most of the computational time of IPOPT ( $> 99\%$ ) is spent in the numerical linear algebra kernel to solve Eq. 9.16 as a solution of a Newton system is needed in every IPOPT iteration. If the problem size is rather small, a direct linear solver is a fast and efficient approach to solve these systems. However, in case of large-scale 3-D PDE-constrained optimization problems a direct linear solver is not an option, while the memory consumption is tremendously high due to the large fill-in, while the cubical time complexity is unacceptable from a practical point of view.

Recently, approximative, *inexact* solutions computed by an iterative solver to solve Eq. 9.16 were incorporated into IPOPT while global convergence is still ascertained [59]. Three termination tests check the acceptability of the search direction based on intermediate solutions of the iterative solver. This offers the opportunity to integrate a parallel iterative solver into the nonlinear optimization process which solves these very large systems efficiently. It is an active research area to design parallel iterative solvers in combination with strong preconditioners to obtain a parallel nonlinear optimization framework.

## Chapter 10

---

# Computational Results

---

In this chapter, first the experimental environment is described, and then PSPIKE, with an emphasis on the role of graph algorithms, and PAUL are benchmarked on various experimental data and data intensive applications.

### 10.1 Experimental Testbed

All performance experiments concerning PSPIKE and the solving of large linear equation systems as, for instance, in applications in arterial fluid dynamics and PDE-constrained optimization, are performed on a Cray XK6. All benchmark results concerning PAUL and solving the weighted bipartite graph matching problem are performed on a Cray XE6.

Both high-performance computing machines are quite similar, but differ in the installed CPUs: AMD Opteron 6172 with 2.1 GHz (“Magny Cours”) in the case of the Cray XE6, whereas the next generation AMD Opteron 6272 with 2.1 GHz (“Interlagos”) is featured in the Cray XK6. Each supercomputer hosts 176 compute nodes; each compute node consists of two compute sockets. The nodes are interconnected through a 3-D torus network. The Gemini communication interface is installed for every two compute nodes. The available memory per node is 32 GB. In the case of the Cray XE6, each socket contains a 12-core AMD Opteron. The Cray XK6 is a hybrid multicore machine, i.e., each node contains a 16-core AMD Opteron and an Nvidia GPU X2090. Both Cray machines are maintained by the Swiss National Supercomputing Centre in Manno,

Switzerland. In all experiments running on the Cray XK6 the GNU programming environment 4.0.30 is used including GCC 4.6.2, while on a Cray XE6 the Pathscale programming environment version 3.1. 61 is employed including the Pathscale compiler 3.2.99.

## 10.2 Benchmark Results with PSPIKE

For the real life matrices of the Florida collection, the crucial role of the solution of the weighted dense  $k$ -subgraph problem in the preprocessing phase of PSPIKE is demonstrated. Then, PSPIKE is applied to linear equation systems obtained by discretizing PDEs in simulation and optimization problems.

### 10.2.1 Florida Collection

In these experiments, the performance of PSPIKE is demonstrated for matrices picked from the *University of Florida Sparse Matrix Collection* [63]. Special attention will be paid to the influence of the solution of the dense  $k$ -subgraph problem on the total solution time and on the number of inner and outer BICGSTAB iterations of PSPIKE. Sparse matrices with a size ranging from 181,343 to 1,489,752 are chosen as the benchmark test suite to conduct first scalability experiments motivated by the fact that these were also selected in [159]. The matrices cover various application areas and differ considerably in the condition number as shown in Table 10.1.

The right-hand side is generated by multiplying the matrix with the vector  $(1, \dots, 1)^T$ . In Figs. 10.1 and 10.2 the following scalability experiments were conducted on a Cray XK6 using at most 128 compute nodes, mapping to each node one MPI process, and setting 16 OpenMP threads per MPI process; in total, up to 2,048 compute cores were used. The left y-axis shows the complete computational time (in seconds), which is used to compute the solution of the linear equation system. The right y-axis displays the number of outer BICGSTAB iterations. In the preprocessing phase, the scaling of the matrix and the permutation of large entries to the diagonal of the matrix are done using MC64 [73]; as implementation for the spectral heuristic MC73 [214] was chosen, and the EA (see Chapter 7) was used to solve the weighted dense  $k$ -subgraph problem. As each candidate block can be treated independently of the other candidate blocks, the EA is called concurrently by the available OpenMP threads.

**Table 10.1:** Structural properties and estimated condition numbers of real life matrices.

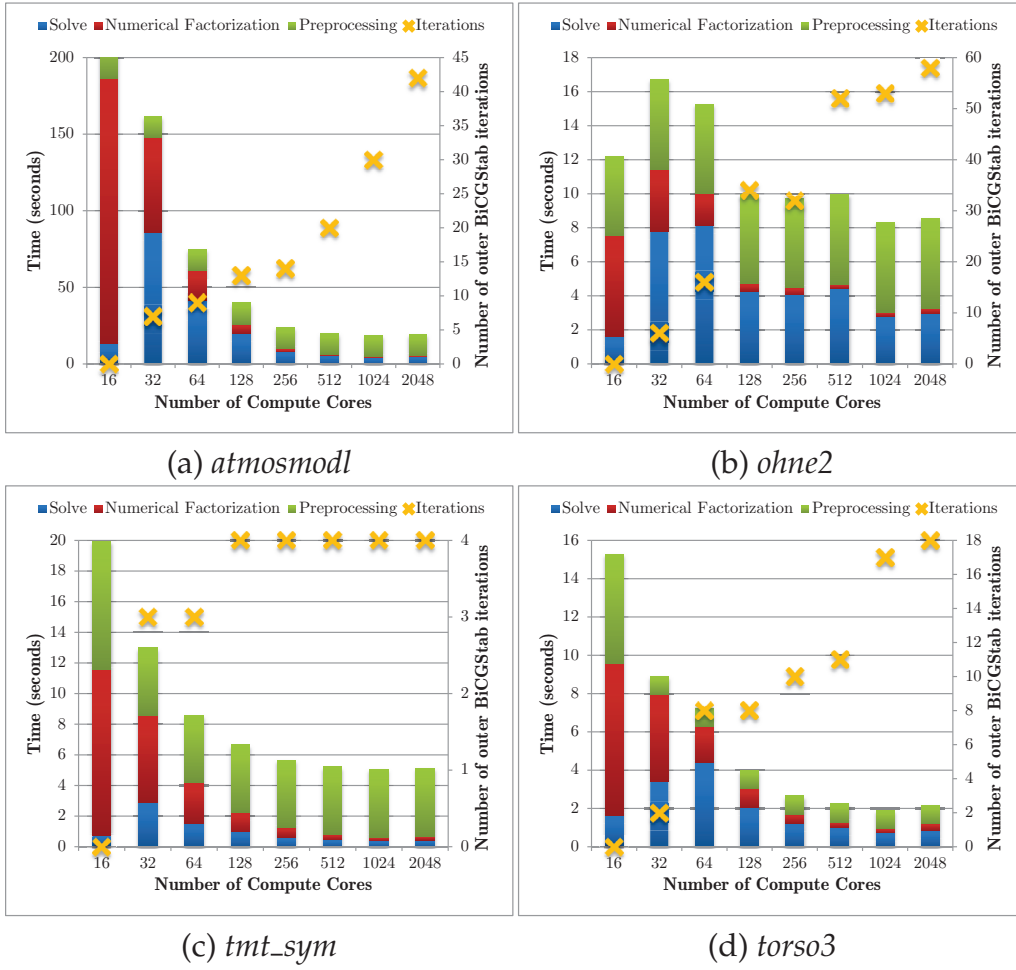
Problem	n	m	m/n	condest	Application
<i>atmosmodl</i>	1,489,752	10,319,760	6.93	1.4728E+03	fluid dynamics
<i>hvdcl2</i>	189,890	1,347,273	7.10	5.5168E+08	power network
<i>language</i>	399,130	1,216,334	3.05	1.2362E+08	weighted graph
<i>ohne2</i>	181,343	11,063,545	61.01	3.5250E+17	semiconductor
<i>thermomech_dk</i>	204,316	2,846,228	13.93	2.2852E+18	thermal
<i>tmt_sym</i>	726,713	2,903,837	4.00	4.1968E+01	electromagnetic
<i>torso3</i>	259,156	4,429,042	17.09	1.3178E+03	2-D/3-D

If PSPIKE is called by one MPI process, then coupling blocks do not exist, and the timings of the parallel direct solver PARDISO using 16 cores are reported in the scalability results. PARDISO factorizes the matrix in the numerical factorization of PSPIKE, and gives an exact solution of the entire linear equation system in the first preconditioner call within the outer BICGSTAB. If more than one MPI process is used, the performance of PSPIKE is measured. The bandwidth is kept to a maximum value of 500. The outer and inner BICGSTAB tolerance levels,  $\epsilon_{\text{out}}$  and  $\epsilon_{\text{in}}$ , are set to  $10^{-5}$  and to, at most,  $10^{-8}$ , respectively. The inner BICGSTAB iteration number is limited to 100 and the maximum number of outer iterations is set to 1,000.

In Fig. 10.1, *atmosmodl* (a), *tmt\_sym* (c), and *torso3* (d) have a rather small estimated condition number; the performance of PSPIKE reflects this fact by showing a very good scaling behavior. The number of outer BICGSTAB iterations depends only weakly on the number of MPI processes, i.e., the number of iterations increases less than linearly with number of processes. A speedup of at most  $50\times$  can be achieved by PSPIKE against PARDISO. Albeit the condition number of matrix *ohne2* (b) is very high, PSPIKE achieves a speedup of  $2\times$  when compared to the performance of PARDISO. In general, for the matrices which tend to be easy to solve, the inclusion of the solution of the weighted dense subgraph has only a negligible influence on the convergence (see, for instance, Fig. 10.2(a)).

Unlike in these rather well-conditioned instances, the solution of the weighted dense  $k$ -subgraph problem dictates the scalability of PSPIKE for ill-conditioned matrices like *hvdcl2* and *thermomech\_dk* as shown in Figs. 10.2(b) and (c). For solving the linear equation system with ma-

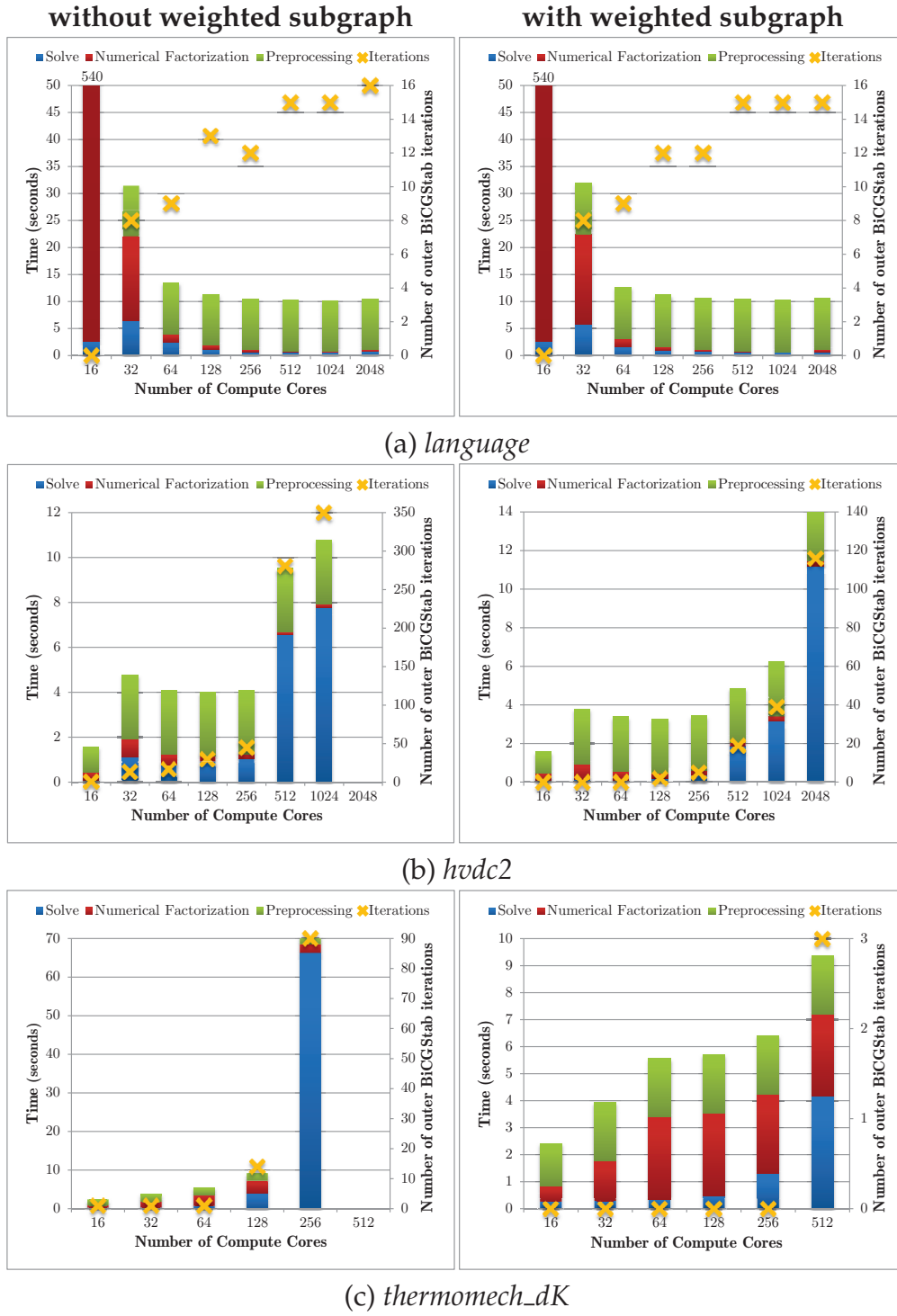




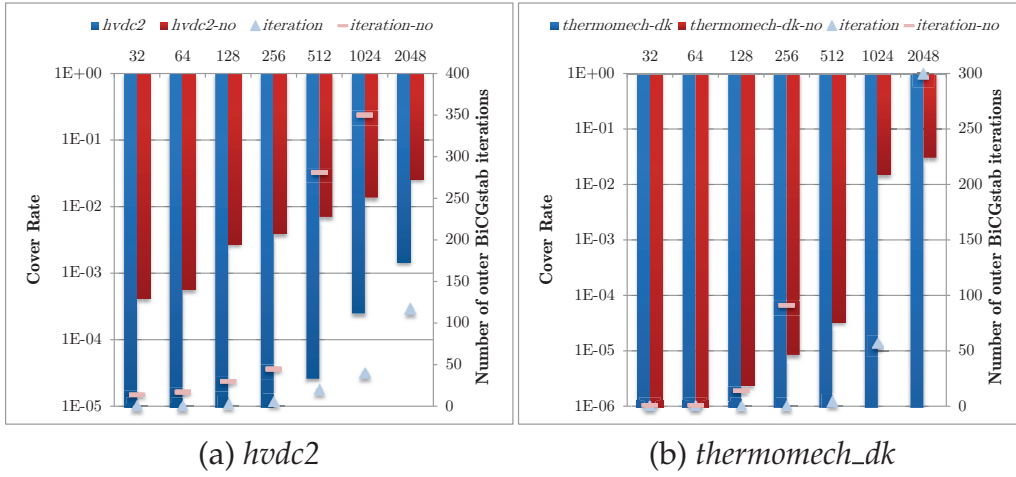
**Figure 10.1:** Scalability results of PSPIKE with up to 2,048 compute cores.

trix *hvd2*, the number of outer BICGSTAB iterations, and thus the total time, are drastically reduced using the reordering of the EA for the weighted dense subgraph problem. If the reordering is omitted, PSPIKE has difficulties converging by using a large number of compute cores. This statement is confirmed by the correlation of the overall cover rate  $\sum_i co_i$  (see Eq. 6.11) to the number of outer BICGSTAB iterations as given in Fig. 10.3. In Figs. 10.3(a) and (b), the left y-axis visualizes the cover rate and the right y-axis indicates the number of outer BICGSTAB iterations until convergence. The x-axis represents the number of compute cores. The smaller the cover rate, the better is the overlapping of the diagonal and coupling blocks. The term “-no” attached to the problem name and to the iteration numbers means that the weighted subgraph problem is not included in this experiment.





**Figure 10.2:** Comparison of scalability results of PSPIKE with and without solving the weighted dense subgraph problem in the preprocessing phase.

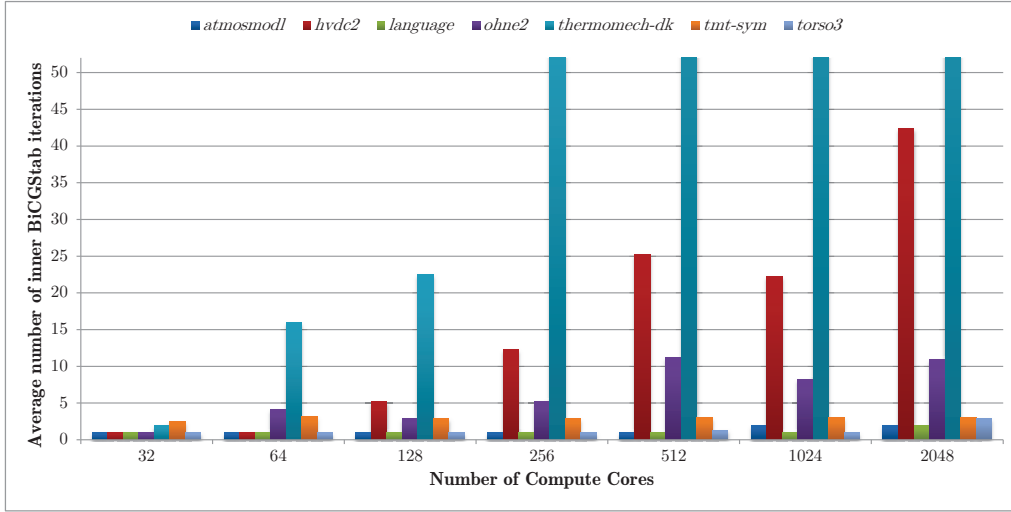


**Figure 10.3:** Relation of the cover rate (with and without including the reordering of the weighted dense subgraph problem) to the corresponding number of outer BICGSTAB iterations.

The overlapping of nonzero elements is almost two orders of magnitude better if the reordering of the weighted dense subgraph problem is included in the preprocessing phase and, consequently, a much better convergence behavior of the full solver can be achieved.

The consequence of not solving the weighted dense subgraph problem in the preprocessing phase is even worse for solving the linear equation system with the matrix *thermomech\_dk*, as it cannot be solved using more than 256 cores as given in Fig. 10.3(b). In the case of including the reordering of the weighted subgraph problem, all nonzero elements are covered by the PSPIKE preconditioner, which indicates that PSPIKE converges in a single iteration if the reduced system (see Eq. 6.5) is solved with a high accuracy. In Fig. 10.4 the average number of inner BICGSTAB iterations for all test problems is shown. For the problem *thermomech\_dk*, the convergence behavior is still obtained using up to 512 compute cores, although the desired accuracy  $\epsilon_{\text{in}}$  of the inner solving step has not been attained. When using more than 512 compute cores, the accuracy of the inner solve is not high enough within 100 inner BICGSTAB iterations (merely  $10^{-1}$  instead of the desired  $10^{-8}$ ), resulting in a large number of outer BICGSTAB iterations.

In PSPIKE, the number of inner BICGSTAB iterations to achieve  $\epsilon_{\text{in}}$  increases with the condition number of the matrix. If the matrix is ill-conditioned, the inner BICGSTAB solve requires more iterations to converge. In general, increasing the number of compute cores will also in-



**Figure 10.4:** Average number of inner BICGSTAB iterations when solving the linear systems.

crease the number of inner and outer BICGSTAB iterations.

The timings (in seconds) of the preprocessing phase are listed for different reordering routines in Table 10.2. The spectral heuristic is the most time-consuming part, while scaling, matching, and weighted subgraph have comparatively low time requirements. Although the acceleration of the spectral heuristic is an important aspect, in real life applications it will not be crucial if the structure of the matrix does not change from iteration to iteration. Then, the solver calls the spectral heuristic only once in the full application.

It can be generally observed that the heavier the coupling block matrices between the diagonal blocks, the faster and more reliably PSPIKE converges. The size of the coupling blocks additionally drops the performance of PSPIKE. The larger the bandwidth  $k$ , the more time the numerical factorization consumes as PARDISO solves systems with  $2k$  right-hand sides. Also the inner BICGSTAB will be more expensive in terms of time as the size of the reduced spike systems is increased. But the positive effect is that the preconditioner in BICGSTAB is more powerful if more weighted entries are covered by the coupling blocks, in which case the convergence rate of the outer BICGSTAB is reduced.

PSPIKE solves sparse linear equations of diverse real life applications while being competitive against the sparse direct solver PARDISO. Since PSPIKE can be successfully evaluated on single linear systems, the solver

**Table 10.2:** *Timings (in seconds) for the preprocessing phase in PSPIKE.*

<b>Problem</b>	<b>Scaling (s)</b>	<b>Spectral (s)</b>	<b>Matching (s)</b>	<b>Weighted Subgraph (s)</b>
<i>atmosmodl</i>	1.14	11.61	0.56	0.62
<i>hvd2</i>	0.17	2.66	0.09	0.07
<i>language</i>	0.17	9.01	0.21	0.15
<i>ohne2</i>	0.78	3.24	0.58	0.68
<i>thermomech_dk</i>	0.29	1.54	0.16	0.17
<i>tmt_sym</i>	0.66	3.25	0.36	0.14
<i>torso3</i>	0.43	-	0.27	0.25

is integrated into two data intensive applications, where a linear system needs to be solved per iteration step, and the solution of the linear system highly influences the overall convergence.

### 10.2.2 Arterial Flow Simulation

Here, a well-known benchmark problem is simulated which describes a flow in a lid-driven square cavity. A commercial parallel fluid solver is used for the simulation of the arterial flow. The solver was developed at the ETH Foundation for Research on Information Technologies in Society (IT<sup>2</sup>S) located in Zurich, Switzerland. In the solver, the PDE is discretized using the finite element method. Then, the solution approach described in Chapter 9.2 is applied, and linear systems (see Eq. 9.10) need to be solved in each Picard iteration. Since the computational time for solving the linear equation systems requires 95% of the overall time, it is mandatory to solve the ill-conditioned, unsymmetric systems efficiently. Currently, a parallel iterative linear system solver consisting of GMRES and a Schur complement preconditioning method is being used, which will be compared to the benchmark tests of PSPIKE.

The objective of using PSPIKE in this application is to demonstrate that the black box solver is also an efficient option to handle such types of problems, but relies not on problem-specific information. In the subsequent experiments, four different mesh sizes (S–XL) were studied. The largest mesh (XL) results in a linear equation system with about 2.5 million unknowns (see Table 10.3). As the fluid solver is implemented using the scientific computation toolkit PETSC [19], PSPIKE was extended by an interface to this toolkit. In the experiments with the meshes S, M, and

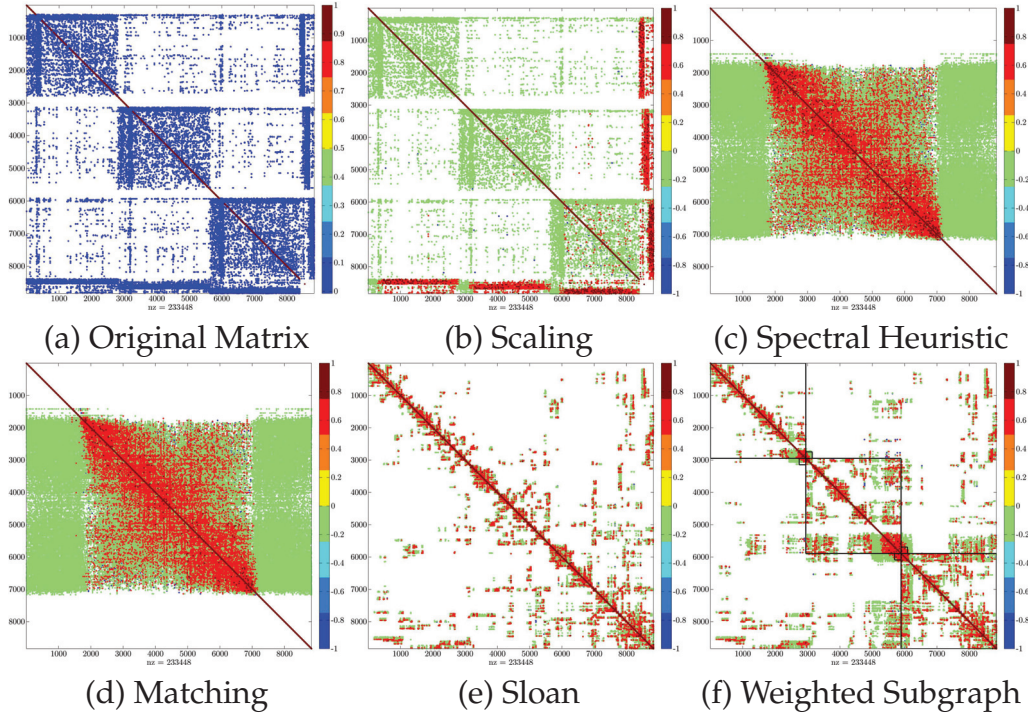
**Table 10.3:** *Mesh and resulting matrix sizes for the arterial flow data.*

Data Set	Cells	Nodes	n	m
S	49,414	71,870	225,436	8,315,125
M	113,871	165,911	520,414	19,013,950
L	223,366	333,262	1,046,231	36,796,912
XL	592,434	858,303	2,692,337	100,930,150

L, the bandwidth in PSPIKE is set to 100, the inner tolerance level to  $10^{-15}$ , and the outer tolerance to  $10^{-6}$ . For the largest mesh XL, the bandwidth is set to 300 and the outer tolerance to  $10^{-7}$ , with the same inner tolerance level as for the smaller meshes.

In Fig. 10.5, the structure of the matrix is visualized which is reordered via the multilevel hybrid partitioning approach (see Eq. 6.10) in each step of the preprocessing phase of PSPIKE (see Sec. 8.3); here demonstrated for three processes. In Fig. 10.5(a), the original structure of the matrix is presented. In the fluid solver, the matrix is already partitioned into the three desired diagonal blocks. However, the coupling blocks are empty and many entries reside on the bottom and right areas of the matrix. The numerical values range from 0 to 1. Large numerical values can be found around the diagonal. In the first preprocessing step (Fig. 10.5(b)), the matrix is scaled to obtain values between  $-1$  and  $1$  using  $2 \times 2$  Match. It can be seen that the values in the bottom and the right areas of the matrix are large, thus most of them need to be covered by either diagonal or coupling blocks in order to create a strong preconditioner in PSPIKE. The spectral heuristic (Fig. 10.5(c)), TRACEMIN-FIEDLER, reduces the bandwidth of the matrix, but also shrinks the matrix. Then, the number of entries per process becomes quite unbalanced when applying the default row-wise block distribution in PSPIKE. The reordering obtained by graph matching (Fig. 10.5(d)) via the implementation MC64 permutes largest entries on the diagonal, and changes the structure of the matrix slightly. In order to further reduce the bandwidth and to improve the balancing of the entries, Sloan's algorithm is applied (Fig. 10.5(e)). Finally, the (1+1)-EA returns a reordering for the weighted subgraph problem (Fig. 10.5(f)) which reorders heavy-weighted entries into the coupling blocks. The resulting diagonal and coupling block matrices are highlighted for a bandwidth of 200.

The computation time for each reordering routine applied to the dif-



**Figure 10.5:** Step-by-step reorderings in the preprocessing phase of PSPIKE to obtain from the original matrix (a) the PSPIKE structure (f) with three processes.

ferent mesh sizes is given in Table 10.4. The spectral heuristic consumes at least 70% of the total time in the preprocessing phase per PSPIKE iteration, here shown when using 256 cores. Despite the implementation being parallel, TRACEMIN-FIEDLER gains a maximum speedup of 2; the time needed by this method still dominates the preprocessing phase. The timings for the scaling method and for Sloan’s algorithm can be an issue for the total solving time, especially when considering large-scale problems. The time required for the weighted matching computation and for the EA to solve the dense weighted subgraph problem can be neglected. The time to compute a reordering depends linearly on the size of the matrix; e.g., the size of the matrix of mesh XL is three times larger than the size of the matrix of mesh L; thus, all reordering routines need three times as much time to compute the reordering.

The initial estimate for the values in the starting solution is set to  $10^{-9}$  in PSPIKE in order to solve the linear equation system in the first Picard iteration. The starting solutions for the other linear systems are set to the solution of the previous iteration. All experiments are performed on the



**Table 10.4:** *Timings (in seconds) of the preprocessing phase in PSPIKE per fluid solver iteration.*

Reordering	Mesh S (s)	Mesh M (s)	Mesh L (s)	Mesh XL (s)
Scaling	2.41	4.44	8.44	26.53
Spectral Heuristic	8.74	22.37	47.16	134.08
Matching	0.49	1.14	2.22	6.42
Sloan	1.96	5.17	10.69	33.10
Weighted Subgraph	0.63	1.27	2.61	7.52

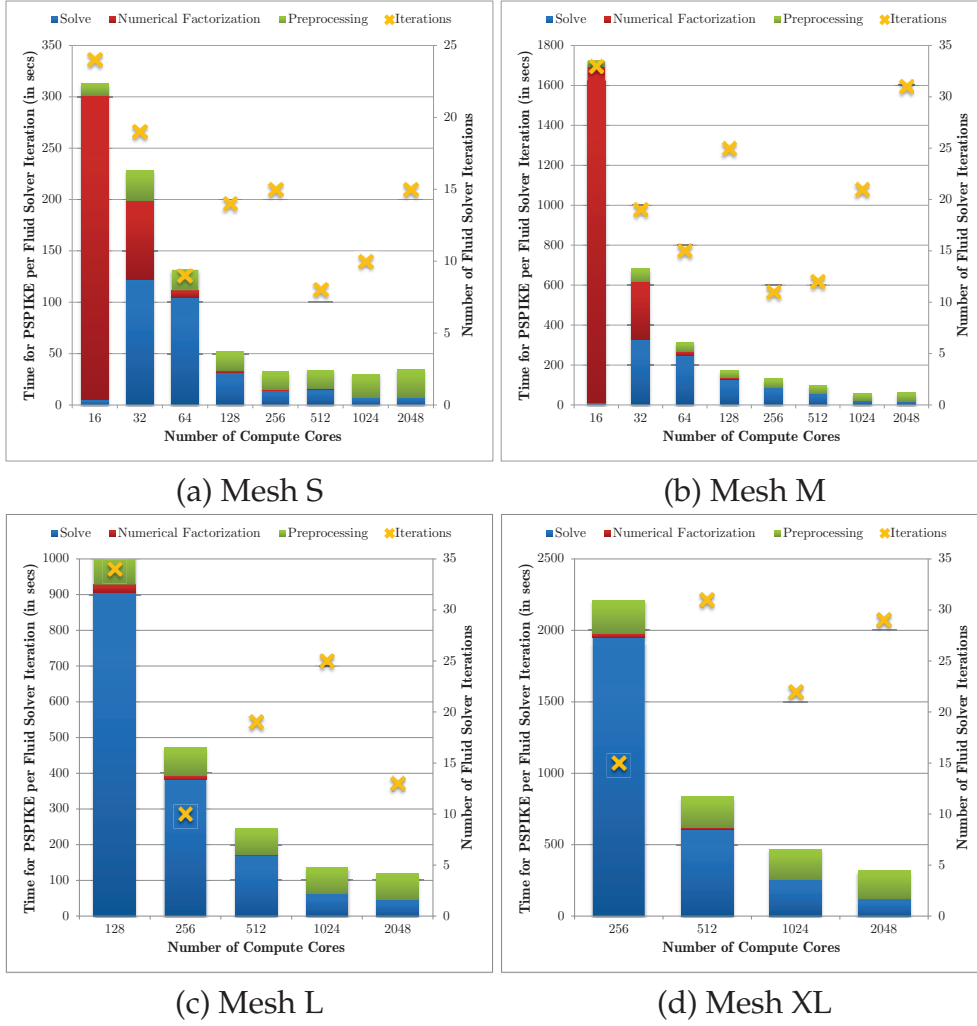
Cray XK6 using up to 2,048 compute cores, with the OpenMP threads always set to 16 on each compute node.

In Fig. 10.6, the performance of PSPIKE per fluid iteration is shown for all the mesh sizes. For the meshes S and M, the first bar indicates the time per fluid iteration of the parallel direct solver PARDISO using 16 OpenMP threads. For the meshes L and XL, the experiments start with using at least 8 and 16 MPI processes, respectively, as PARDISO requires more memory than is available on a node. Otherwise, PSPIKE will require more than the upper limit of 12 hours for the entire simulation process.

In PARDISO, most of the time is spent on the numerical factorization of the matrix while in PSPIKE most of the time is spent in the solving phase. The time for the numerical factorization in PSPIKE can be neglected due to the fact that the size of the diagonal block matrices becomes smaller as more compute cores are used. When increasing the number of compute cores up to 2,048, the preprocessing time and, in particular, the time required for the spectral heuristic, becomes the dominant part in the computation.

The x-shaped markers in Fig. 10.6 show that the direct sparse linear solver needs more fluid iterations to converge to the tolerance level than PSPIKE. In general, the number of fluid iterations is in the range of 8 to 34. It can be concluded that the fluid solver is not mesh independent since different numbers of fluid iterations are performed when changing the mesh size but using the direct linear solver PARDISO. As PSPIKE does not return the same solution of the linear equation system when increasing the number of compute cores, the number of fluid iterations does not stay constant.

In Fig. 10.7, the speedups and the parallel efficiencies for the com-

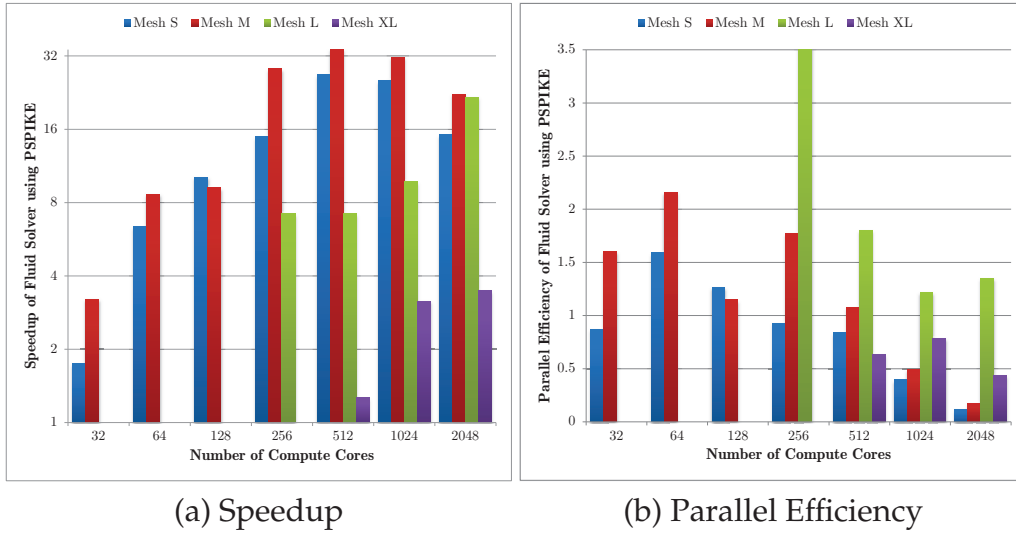


**Figure 10.6:** Time of PSPIKE per fluid iteration and total number of fluid iterations using PSPIKE for the different mesh sizes.

plete simulation process of the fluid solver using PSPIKE are shown. The speed improvement against PARDISO is shown for meshes S and M, while for the meshes L and XL the speed improvement of PSPIKE against itself with 128 and 256 compute cores, respectively, is presented in Fig. 10.7(a). Although the number of fluid iterations is not constant when using different numbers of compute cores, a speedup can still be obtained for all mesh sizes. A maximum speedup of 32 was observed when comparing the performance of PSPIKE to PARDISO using 512 compute cores. When using 2,048 compute cores, the reordering time dominates the overall time and drops the speedup number.

The parallel efficiency of PSPIKE in the fluid solver is illustrated in



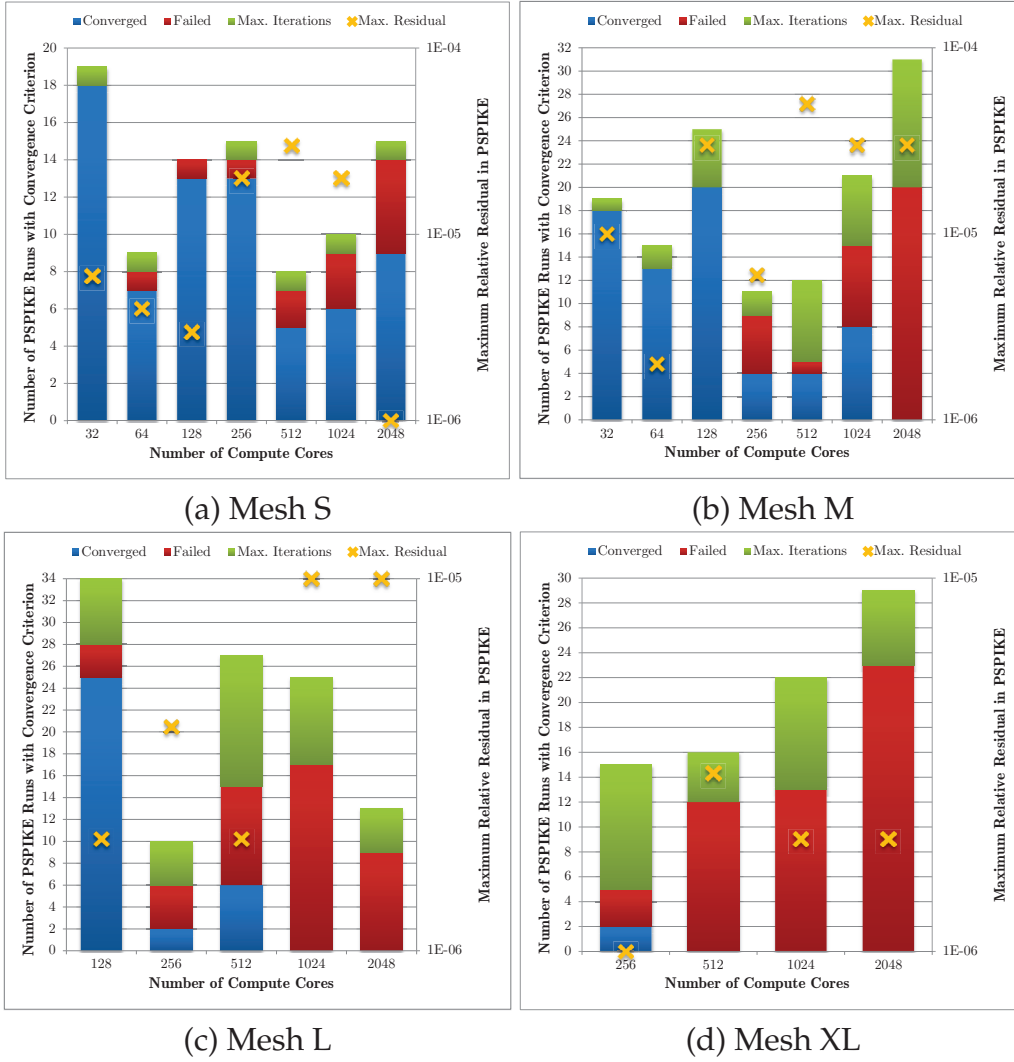


**Figure 10.7:** The speedup and parallel efficiency for the complete simulation process when using PSPIKE in the fluid solver.

Fig. 10.7(b). The efficiency dropped below 1 for the smaller meshes S and M when more than 512 compute cores were used. In case of the mesh L, the parallel fluid solver is quite efficient when using up to 2,048 compute cores, while for mesh XL, the efficiency is small when the maximum number of compute cores is used due to the dominant preprocessing time. An efficiency number greater than 1 is achieved due to the fact that the number of fluid iterations is not constant and the time to converge may decrease drastically.

There are three possible stopping criteria of PSPIKE: in the usual case, PSPIKE will converge to the desired relative residual tolerance and will return the corresponding solution of the linear equation system. These successful runs are shown by the blue bars labeled “Converged.” However, it might also happen that the iterative solver BICGSTAB fails and is not able to improve the best solution found so far. The bars corresponding to this scenario are labeled “Failed” (in red). The last stopping criterion, “Max. Iterations,” which is shown in green, means that the maximum number of BICGSTAB iterations has been reached; here, the maximum number was set to 500. In both failure situations, PSPIKE returns the best solution found so far, which is in the worst case a solution with an accuracy on the order of  $10^{-5}$ .

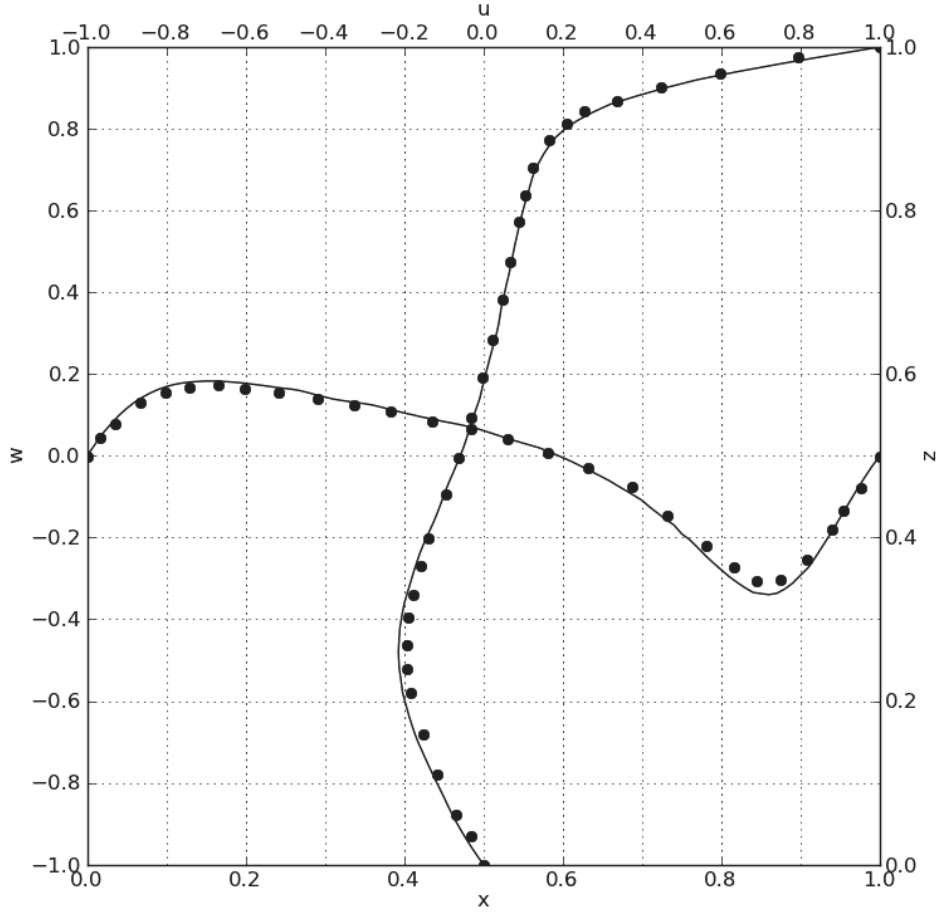
The failure situations occur most commonly when the preconditioner is not strong enough for the particular matrix. Certainly, the strength



**Figure 10.8:** Possible convergence criteria of PSPIKE, number of their occurrences, and reached maximum residuals.

of the preconditioner in PSPIKE can be enhanced by choosing a larger bandwidth, but it will impact the time needed for solving the linear equation system; and as long as the full simulation process converges to the desired residual tolerance the chosen configuration is the fastest option.

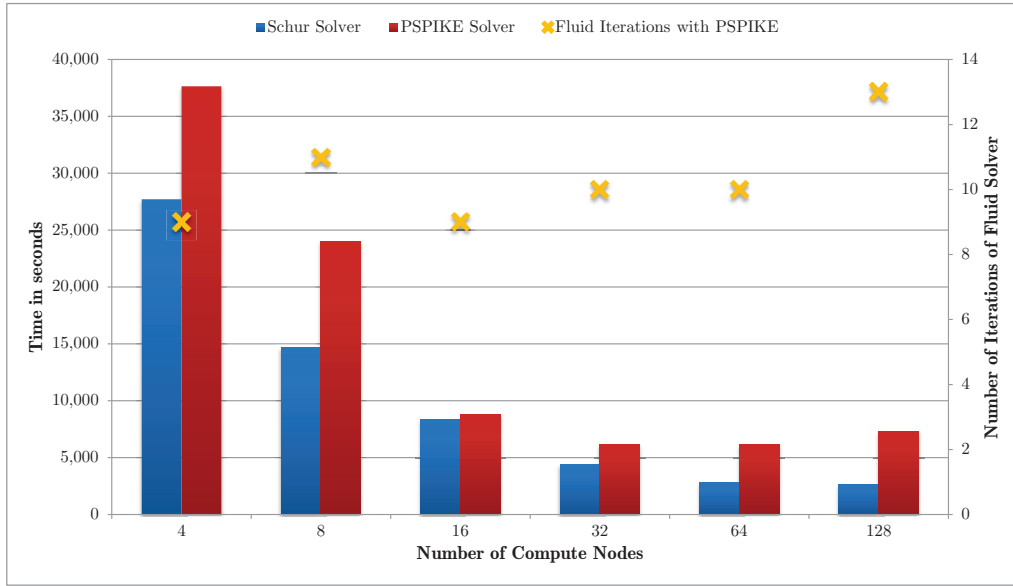
In Fig. 10.8, the total number of fluid iterations is illustrated by the bars on the left vertical axis, which are split into the three stopping criteria: maximum number of iterations reached, iterative solver failed, and accuracy of the relative residual is small enough. The right vertical axis displays the maximum relative residual returned by PSPIKE indicated by the x-shaped markers. In the mesh S dataset (Fig. 10.8(a)), PSPIKE



**Figure 10.9:** Comparison of the  $u$  and  $w$  velocity profiles at the plane of symmetry  $y = 0.5$ .

converges to the desired accuracy in most of the cases. But as more compute cores are involved in the computation, the failure rate increases. However, the relative residual is still good enough for the fluid solver. For the other meshes (Figs. 10.8(b)–(d)), the failure situation appears more often with the PSPIKE bandwidth being set to 100, which results in a lower cover rate, but in a faster convergence of the solver.

The velocity profile of the fluid solver combined with PSPIKE is compared to the velocity profile given in [102] for the mesh M (see Fig. 10.9). The dotted points are the reference solutions for the  $u$  and  $w$  components of the velocity vector in the  $z$  and  $x$  directions, respectively. The continuous line is the solution computed by the fluid solver with PSPIKE. It



**Figure 10.10:** Comparison between the Schur Solver and PSPIKE for the mesh dataset XL.

can be concluded that both vector components can be recovered highly efficiently by using PSPIKE in the fluid solver, and an almost identical solution quality is achieved compared to the reference data.

Eventually, the performance of PSPIKE for solving the linear equation systems is compared to the existing distributed-memory linear solver in the fluid solver, called “Schur solver,” for the mesh XL dataset. The timings (in seconds) of both parallel solvers are obtained as follows: given a fixed number of compute nodes, the fastest running time of the entire simulation process is measured when using PSPIKE and when using the Schur solver.

The fastest results for PSPIKE are obtained by mapping at most two MPI processes on a compute node, and setting the OpenMP threads per MPI process to at least 8. The bandwidth in PSPIKE is fine tuned according to the number of compute nodes employed and ranges between 300 and 600. Only MPI processes are involved in the computation with the Schur solver and its best performance is achieved by mapping each MPI process to a compute node. The experiments are repeated several times and the best timing result is taken as the final running time.

In Fig. 10.10, a comparison of timings and overall iterations between the two solvers are presented. The number of compute nodes is plotted on the x-axis, the left y-axis displays the time in seconds for the entire

simulation process, and the right y-axis visualizes the number of fluid iterations when using PSPIKE. The performance scalability of PSPIKE is comparable to the scalability of the Schur solver when using up to 32 nodes, corresponding to 512 compute cores. When using more nodes for PSPIKE, the number of fluid iterations almost doubles which results in a large computational time for PSPIKE, and thus for the entire simulation process. The Schur solver always needs 47 fluid iterations to converge to the accuracy level. This number is achieved independently of the number of nodes employed.

The flexibility and scalability of the hybrid solver have been demonstrated in the benchmark tests. The PSPIKE solver does not need any problem specific input, but it still converges efficiently to an acceptable solution. The solution quality and performance scalability are comparable to a fine tuned, problem specific, and scalable Schur solver.

The performance of the fluid solver with PSPIKE can be further improved by incorporating a convergence check module into the fluid solver. This module would check if the accuracy of the solution for the current linear equation system is low enough. Thus, the control over the convergence of the linear solver would be given to the fluid solver, which evaluates the solution quality with respect to the search direction and the step length for the next Picard iteration. This would also help achieve a constant number of fluid iterations.

A further part which needs improvement is the robustness of the fluid solver when using PSPIKE: it is currently not possible to repeat a run of the fluid solver subject to getting identical results. Although a identical setup of the last run has been used and the solver PSPIKE returns the identical solution even in parallel (when disabling OpenMP threading). The situation might be even worse as the number of fluid iterations does not remain constant per experiment which clearly influences the performance scalability of PSPIKE. A major reason for the instability of the fluid solver is due to the fact that PETSC in parallel does not generate bitwise identical results because of rounding errors [66]. Unfortunately, this accuracy error is already large enough to bias the entire solving process including the fluid solver and PSPIKE.

In summary, it cannot be expected that the performance of the “black-box” solver PSPIKE dominates the best problem-specific implementation since these solvers are adapted to the specific problem structure. But the benchmarks illustrate that PSPIKE is competitive with one of the fastest implementation due to the essential role of graph algorithms.

### 10.2.3 Optimal Control of Partial Differential Equations

Within the nonlinear optimization framework IPOPT, large-scale nonlinear optimization problems have been implemented which are the basis for benchmarking PSPIKE. Two PDE-constrained optimization problems have been selected for strong scalability tests: a nonconvex and a convex 3-D optimal control problem. The former problem [165, 211], a *boundary control* problem, is formulated as

$$\begin{aligned} \min_{y,u} \quad & \int_{\Omega} \Phi(y(\mathbf{x}) - y_t(\mathbf{x})) d\mathbf{x} + \alpha \int_{\partial\Omega} u(\mathbf{x})^2 d\mathbf{x} \\ \text{s.t.} \quad & -\Delta y(\mathbf{x}) = y_s \quad \text{on } \Omega, \\ & y(\mathbf{x}) \leq y_r \quad \text{on } \Omega, \\ & y(\mathbf{x}) = u(\mathbf{x}) \quad \text{on } \partial\Omega, \\ & u_l \leq u(\mathbf{x}) \leq u_r \quad \text{on } \partial\Omega, \end{aligned}$$

where  $\alpha = 10^{-2}$ ,  $y_s = 20$ ,  $y_r = 3.2$ ,  $u_l = 1.6$ ,  $u_r = 2.3$ ,  $y, y_t, u : \mathbb{R}^3 \rightarrow \mathbb{R}$ ,  $\Omega = [0, 1]^3$ , the target profile  $y_t(\mathbf{x}) = c_1 + c_2 \cdot x_1(x_1 - 1) \cdot x_2(x_2 - 1) \cdot x_3(x_3 - 1)$  with  $c_1 = 2.8$ ,  $c_2 = 40$ , and  $\Phi(\mathbf{x})$  is the Beaton–Tukey penalty function.

The objective function minimizes the norm — computed via  $\Phi(\mathbf{x})$  — to a target function  $y_t$ , while the second part in the objective function is the Tikhonov regularization term which ensures the well-posedness of the problem. The equality constraints include the PDE, and the inequality constraints are the lower and upper bounds on the control and state functions, respectively. On the boundary of the 3-D domain  $\Omega$ , Dirichlet boundary conditions are imposed, i.e., the control function bounds the state function with respect to lower and upper bounds.

The latter convex optimization problem [166], a *distributed control* problem, is posed as

$$\begin{aligned} \min_{y,u} \quad & \int_{\Omega} (y(\mathbf{x}) - y_t(\mathbf{x}))^2 d\mathbf{x} + \alpha \int_{\partial\Omega} u(\mathbf{x})^2 d\mathbf{x} \\ \text{s.t.} \quad & -\Delta y(\mathbf{x}) - y(\mathbf{x}) + y(\mathbf{x})^3 = u(\mathbf{x}) \quad \text{on } \Omega \\ & u_l \leq u(\mathbf{x}) \leq u_r \quad \text{on } \Omega, \\ & y(\mathbf{x}) \leq y_r \quad \text{on } \Omega, \\ & y(\mathbf{x}) = y_s \quad \text{on } \partial\Omega, \end{aligned}$$

where  $\alpha = 10^{-3}$ ,  $y_s = 0$ ,  $y_r = 0.185$ ,  $u_l = 1.5$ ,  $u_r = 4.5$ ,  $y, y_t, u : \mathbb{R}^3 \rightarrow \mathbb{R}$ ,  $\Omega = [0, 1]^3$ ,  $y_t(\mathbf{x}) = c_1 + c_2 \cdot (x_1(x_1 - 1) + x_2(x_2 - 1) + x_3(x_3 - 1))$  with

**Table 10.5:** Size of control problems, PSPIKE parameters bandwidth  $k$  and inner tolerance level  $\epsilon_{\text{in}}$ , and maximum IPOPT iterations  $it$ .

Problem	N	n	m	k	$\epsilon_{\text{in}}$	it
Boundary Control	100	4,300,000	62,772,832	100	$10^{-8}$	11
	150	14,175,000	210,994,232	200	$10^{-12}$	11
Distributed Control	80	4,608,000	18,867,200	200	$10^{-8}$	8
	110	11,979,000	49,101,800	500	$10^{-12}$	7

$c_1 = 1$ ,  $c_2 = \frac{4}{3}$ . Here, the PDE is controlled over the entire 3-D domain by the control function  $u$ ; upper and lower bounds are also included for both functions. Again, Dirichlet boundary conditions are applied on the boundary of the domain.

These two large-scale PDE-constrained optimization problems are discretized via finite differencing by setting the discretization parameter  $N$  to obtain a medium-sized and a large-sized Newton system: in the case of boundary control  $N = \{100, 150\}$  and in the case of distributed control  $N = \{80, 110\}$  (cf. Table 10.5). In PSPIKE, the bandwidth  $k$  and the inner tolerance level  $\epsilon_{\text{in}}$  are adapted to the size of the problem. In order to be able to compare the total timings for the full optimization process\* by increasing the number of cores, the number of IPOPT iterations is set to a maximum value.

Typically, 99% of the overall optimization time is consumed by the linear solver solving the Newton system (see Eq. 9.16).

As the structure of the Newton system does not change during the optimization, the spectral heuristic is called once in the preprocessing phase of PSPIKE, and the same reordering is applied in all subsequent IPOPT iterations. The other reordering techniques are computed in every IPOPT iteration. Accumulated timings (in seconds) over all IPOPT iterations are provided for the four reordering routines in Table 10.6. For the scaling,  $2 \times 2$  Mat is used; for the spectral reordering, MC73; for the matching, MC64; and for the weighted subgraph problem, the (1+1)-EA (see Sec. 8.3). The time required for computing the spectral reordering is still higher than the time required of other reordering routines, but the time difference is less dramatic.

The resulting state and control functions of the PDE-constrained op-

\*In the current parallel version 3.9.2 of IPOPT, PSPIKE is enabled by setting the option “*with\_pspike yes*” in the IPOPT’s option file *ipopt.opt*.



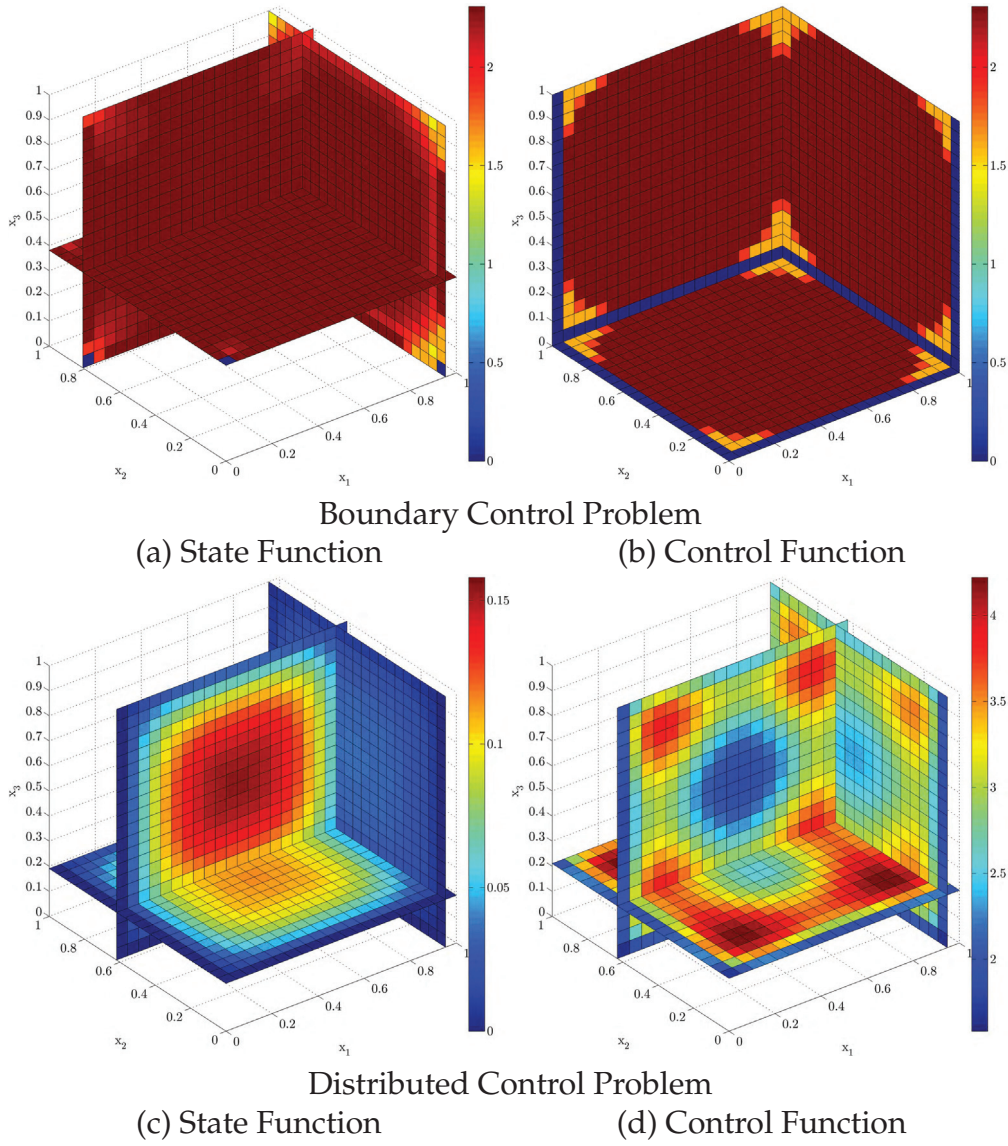
**Table 10.6:** *Accumulated timings (in seconds) of the preprocessing phase in PSPIKE for the full optimization process.*

Problem	N	Scaling (s)	Spectral (s)	Matching (s)	Weighted Subgraph (s)
Boundary Control	100	72.43	142.84	44.21	38.87
	150	217.48	475.39	148.05	111.12
Distributed Control	80	27.48	89.06	17.40	12.70
	110	55.54	210.83	37.68	23.85

timization problems are presented in Fig. 10.11. In all figures, the state and control functions are visualized by slice planes considering the dimensions  $x_1, x_2, x_3$  in  $\Omega = [0, 1]^3$ . The magnitude of the values of the functions are illustrated by colors ranging from “dark red” for a large value to “dark blue” for a small value. In Fig. 10.11(a), the values of the state function are truncated in directions to the corners of the boundaries. The control function, as shown in Fig. 10.11(b), has exactly the value zero at the boundaries. For the distributed control problem, the state function is bell shaped as illustrated in Fig. 10.11(c); values decay towards the boundaries. The control function, shown in Fig. 10.11(d) is activated on the interior of the PDE.

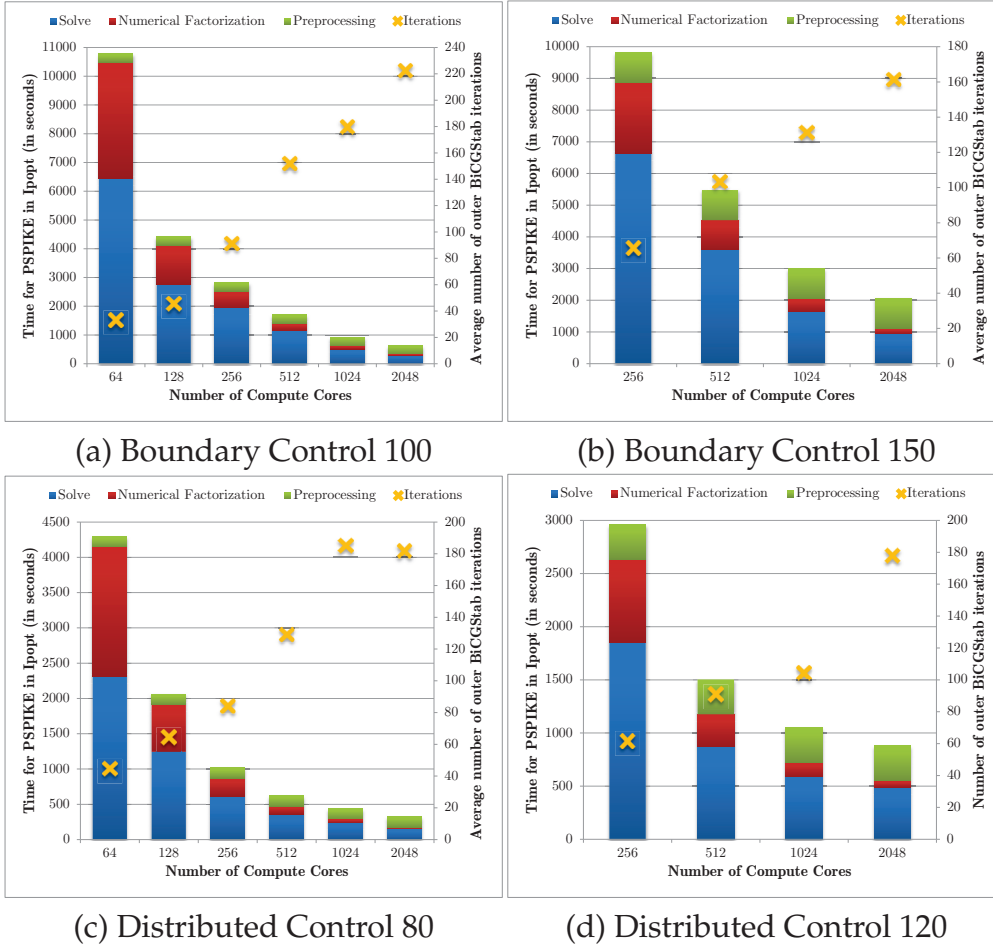
The parallel nonlinear optimization framework is benchmarked on the Cray XK6 using at most 2,048 compute cores. The scalability of PSPIKE is illustrated in Fig. 10.12. The accumulated time over all IPOPT iterations is plotted against the average number of outer BICGSTAB iterations within IPOPT. For the timings of PSPIKE, the time consumed for the convergence check, i.e., the communication of the current solution of PSPIKE to IPOPT and the acceptability tests for the solution, is neglected. Doubling the number of compute cores nearly linearly increases the average number of outer BICGSTAB iterations. This observation can be explained: every doubling of the number of processes has the effect that the diagonal and coupling blocks cover fewer of the weighted entries. Thus, more outer iterations are needed for convergence to an acceptable solution. On average the number of outer BICGSTAB iterations increases by 30% when the number of compute cores is doubled. By using 2,048 compute cores, the preprocessing time in PSPIKE requires almost 50% of the total time, which will be a scalability issue when going beyond 128 MPI processes.





**Figure 10.11:** *Solution of the state and control functions for the PDE-constrained optimization problems.*

In summary, the integration of PSPIKE into the nonlinear optimizer IPOPT constructs a parallel nonlinear optimization framework which tackles large-scale nonlinear 3-D PDE-constrained optimization problems. Thereby, graph algorithms reorder the KKT systems in such a way that PSPIKE and also IPOPT converge to an acceptable solution. In order to enhance the robustness and convergence of PSPIKE for KKT systems, a well-proven preconditioner for these systems could also be handed over to PSPIKE; then, a reordering scheme is extracted from the precondi-



**Figure 10.12:** Timings (in seconds) and average number of iterations for PSPIKE within IPOPT to solve large-scale PDE-constrained optimization problems.

tioned system, and this reordering is applied to the original KKT system which might better take into account the numerical properties of the matrix.

### 10.3 Benchmark Results with PAUL

Driven by the need for solving the maximum weighted matching problem in graph similarity and in parallel sparse linear solvers [73, 206, 210], the quality and scalability of the PAA and the  $\epsilon$ -PAA are compared to state-of-the-art sequential weighted matching implementations. The implementation of the parallel auction algorithm PAUL follows the hybrid

**Table 10.7:** *Structural properties of real life matrices.*

<b>Problem</b>	<b>n</b>	<b>m</b>	<b>m/n</b>	<b>Application</b>
<i>af_shell10</i>	1,508,065	27,090,195	17.93	structural
<i>atmosmodj</i>	1,270,432	8,814,880	6.94	fluid dynamics
<i>audikw_1</i>	943,695	77,651,847	82.29	structural
<i>av41092</i>	41,092	1,683,902	40.98	2-D/3-D
<i>barrier2-4</i>	113,076	3,805,068	33.65	semiconductor device
<i>bmw3_2</i>	227,362	5,757,996	25.33	structural
<i>bone010</i>	986,703	36,326,514	36.81	model reduction
<i>cage15</i>	5,154,859	99,199,551	19.24	weighted graph
<i>circuit5M</i>	5,558,326	59,524,291	10.71	circuit simulation
<i>crankseg_2</i>	63,838	7,106,348	111.32	structural
<i>Freescalc1</i>	3,428,755	18,920,347	5.52	circuit simulation
<i>g7jac020</i>	5,850	45,465	7.77	economic
<i>g7jac020sc</i>	5,850	45,465	7.77	economic
<i>Ge99H100</i>	112,985	4,282,190	37.90	quantum chemistry
<i>Hamrle3</i>	1,447,360	5,514,242	3.81	circuit simulation
<i>helm2d03</i>	392,257	1,567,096	4.00	2-D/3-D
<i>hood</i>	220,542	5,494,489	24.91	structural
<i>human_gene1</i>	22,283	12,345,963	554.05	weighted graph
<i>ibm_matrix_2</i>	51,448	1,056,610	20.54	semiconductor device
<i>matrix_9</i>	103,430	2,121,550	20.51	semiconductor device
<i>mouse_gene</i>	45,101	14,506,196	321.64	weighted graph
<i>ncvxp5</i>	62,500	424,966	6.80	optimization
<i>nd24k</i>	72,000	14,393,817	199.91	2-D/3-D
<i>rajat31</i>	4,690,002	20,316,253	4.33	circuit simulation
<i>scircuit</i>	170,998	958,936	5.61	circuit simulation
<i>Si41Ge41H72</i>	185,639	7,598,452	40.93	quantum chemistry
<i>thermal2</i>	1,228,045	4,904,179	3.99	thermal
<i>torso1</i>	116,158	8,516,500	73.32	2-D/3-D
<i>TSOPF_RS_b678</i>	35,696	8,781,949	246.02	power network

MPI–OpenMP paradigm where each MPI process receives a part of the bipartite graph and runs the bidding phase via OpenMP threading. The benchmark results were obtained on a Cray XE6 using, at maximum, 1,024 compute cores. The combination of processes  $P$  and threads  $T$  with the minimal computational time is taken as the final time for the specific number of cores. For instance, the minimal time for 4 compute cores is taken over the three runs  $(P, T) = \{(1, 4), (2, 2), (4, 1)\}$ . In all performance tests, the time to distribute the matrix is omitted.

### 10.3.1 Sparse Linear Algebra

The performance of PAUL including the two variants PAA and  $\epsilon$ -PAA (see Chapter 4) is compared to the quality and time of existing weighted graph matching implementations like the state-of-the-art sequential implementation MC64, a simple greedy heuristic (see Algorithm 2.1), and an approximation algorithm (“approx”) [163]. All algorithms are benchmarked on a selected set of 29 matrices from the *University of Florida Sparse Matrix Collection* [63] which are mapped to the bipartite graphs representation.

The characteristics and sparsity metrics of the matrices are presented in Table 10.7. In the computational results, the objective function  $W_2 = \prod_{(i,j) \in M} |a_{ij}|$  is maximized and the time is measured in seconds, as given in Table 10.8. In this table, the second column displays the logarithm of the value of the objective function  $W_2$ . The first entry in each algorithm-specific column shows the percentages of the maximum weights achieved by the respective algorithm. For the matrices *g7jac020*, *Hamrle3*, and *TSOPF\_RS\_b678\_c2*, the greedy and approximation algorithm attain an opposite sign for the weight compared to the optimal weight, and thus the value of the weight is given explicitly. Below the weight the time in seconds required to compute the matching can be found. The numbers in brackets represent either the percentage of how many edges have been matched in relation to number of edges in the maximum matching (greedy, approx), or the number of iterations in the auction algorithm (PAA). In cases of the greedy and approximation algorithms, matchings with a high weight are obtained. In all our experiments, MC64 and PAA find a perfect matching, but the latter will not necessarily achieve a matching with an optimal weight due to the influence of the  $\epsilon$ -scaling mechanism.

In many cases, the simple greedy algorithm is able to find a perfect weighted matching in a reasonable time. However, there are some instances in which the greedy algorithm only attains a maximal matching (see, e.g., *ncvxqp5* or *g7jac020*). The quality of the matching attained by the approximation algorithm is comparable to or even better than the matching of the greedy algorithm, but using only a small fraction of the time required by the greedy algorithm.

In the greedy algorithm, the sorting of the edges dominates the overall time (e.g., *audikw\_1* or *bone010*), which can be avoided when using, for instance, the approximation algorithm. Then, a speedup of almost  $2\times$

**Table 10.8:** Comparison of the speed and quality of the greedy heuristic, MC64, and the PAA for real life sparse matrices.

Problem	$\log(W_2)$	Greedy (Match)	Approx (Match)	MC64	PAA (#it)
<i>af_shell10</i>	17,743,020	100% (100%) 13.92 s	100% (100%) 1.04 s	100% 1.76 s	100% (3) 1.91 s
<i>atmosmodj</i>	14,571,427	100% (100%) 3.33 s	100% (100%) 0.40 s	100% 0.67 s	100% (2) 0.81 s
<i>audikv_1</i>	13,573,470	99.92% (99.99%) 42.26 s	100% (100%) 1.31 s	100% 4.43 s	99.99% (27) 4.67 s
<i>av41092</i>	-9,573	34.45% (91.73%) 0.67 s	54.20% (95.03%) 0.14 s	100% 5.09 s	99.82% (42,652) 0.44 s
<i>barrier2-4</i>	1,022,162	61.47% (99.87%) 1.44 s	99.99% (100%) 0.19 s	100% 0.20 s	100% (26) 0.44 s
<i>bmw3_2</i>	2,893,855	99.95% (99.96%) 2.76 s	100% (100%) 0.26 s	100% 0.38 s	100% (2,617) 0.52 s
<i>bone010</i>	9,453,191	100% (100%) 14.54 s	100% (100%) 0.95 s	100% 1.85 s	100% (2) 2.81 s
<i>cage15</i>	-2,000,600	100% (100%) 41.80 s	100% (100%) 4.42 s	100% 6.68 s	100% (2) 6.85 s
<i>circuit5M</i>	-64,084,172	99.99% (100%) 25.16 s	100.01% (99.99%) 2.32 s	100% 4.21 s	100% (6) 4.22 s
<i>crankseg_2</i>	1,003,866	100% (100%) 3.55 s	100% (100%) 0.22 s	100% 0.41 s	100% (4) 0.42 s
<i>Freescape1</i>	-16,578,958	85.86% (100%) 8.15 s	100% (100%) 1.29 s	100% 1.55 s	100% (6) 1.74 s
<i>g7jac020</i>	-8,483	1,628 (86.09%) 0.01 s	1,628 (85.50%) 0.002 s	100% 0.03 s	100% (469) 0.01 s
<i>g7jac020sc</i>	3,900	126.25% (87.38%) 0.01 s	148.48% (86.89%) 0.002 s	100% 0.04 s	100% (2,239) 0.01 s
<i>Ge99H100</i>	294,675	100% (100%) 1.74 s	100% (100%) 0.13 s	100% 0.25 s	100% (2) 0.26 s
<i>Hamrle3</i>	-1,078,904	30,572 (83.68%) 1.96 s	-21.748 (83.32) 0.57 s	100% 209.30 s	99.92% (2,638,082) 14.66 s
<i>helm2d03</i>	501,026	100% (100%) 0.57 s	100% (100%) 0.11 s	100% 0.15 s	100% (2) 0.16 s
<i>hood</i>	2,922,957	100% (100%) 2.56 s	100% (100%) 0.2 s	100% 0.33 s	100% (4) 0.37 s
<i>human_gene1</i>	0	100% (100%) 6.55 s	100% (100%) 0.27 s	100% 0.68 s	100% (2) 0.69 s
<i>ibm_matrix_2</i>	-914,416	99.69% (100%) 0.37 s	97.30% (100%) 0.06 s	100% 0.06 s	100% (27) 0.09 s
<i>ibm_matrix_2_trans</i>	-914,416	92.21% (100%) 0.37 s	97.30% (100%) 0.07 s	100% 0.62 s	100% (413) 0.51 s
<i>matrix_9</i>	-2,064,252	99.65% (99.88%) 0.78 s	98.12% (100%) 0.13 s	100% 0.13 s	100% (822) 0.17 s
<i>matrix_9_trans</i>	-2,064,252	94.67% (100%) 0.78 s	98.12% (100%) 0.14 s	100% 0.80 s	100% (88) 0.57 s
<i>mouse_gene</i>	0	100% (100%) 6.91 s	100% (100%) 0.32 s	100% 0.81 s	100% (2) 0.82 s
<i>ncvxp5</i>	425,120	116.47% (83.50%) 0.14 s	124.50% (80%) 0.04 s	100% 1.32 s	99.98% (6,930) 0.52 s
<i>nd24k</i>	426,379	99.96% (99.95%) 7.13 s	100% (100%) 0.42 s	100% 0.82 s	100% (16) 0.86 s
<i>rajat31</i>	1,094,658	100.02% (99.97%) 8.29 s	100.02% (99.97%) 1.20 s	100% 1.80 s	100% (7,510) 1.92 s
<i>scircuit</i>	-187,229	100.01% (99.99%) 0.37 s	100.03% (99.99%) 0.06 s	100% 0.09 s	100% (79) 0.10 s
<i>Si41Ge41H72</i>	563,430	100% (100%) 3.32 s	100% (100%) 0.22 s	100% 0.45 s	100% (2) 0.46 s
<i>thermal2</i>	1,639,077	100% (100%) 2.45 s	100% (100%) 0.37 s	100% 0.50 s	100% (2) 0.54 s



**Table 10.8:** Comparison of the speed and quality of the greedy heuristic, MC64, and the PAA for real life sparse matrices (continued).

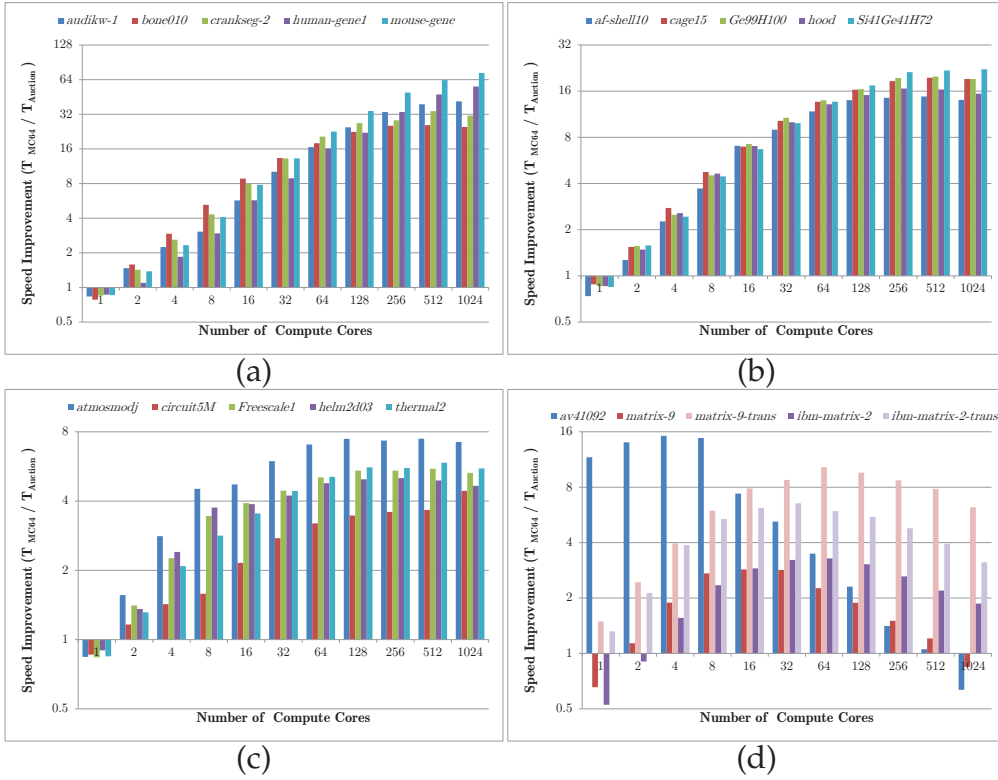
Problem	$\log(W_2)$	Greedy (Match)	Approx (Match)	MC64	PAA (#it)
<i>torso1</i>	388,399	99.80% (99.99%) 3.65 s	99.79% (100%) 0.87 s	100% 4.95 s	99.99% (1,370) 4.66 s
<i>TSOPF_RS_b678_c2</i>	7,359	-153,543 (99.89%) 3.81 s	-150,588 (99.28%) 0.47 s	100% 0.84 s	99.99% (13,222) 0.83 s

against MC64 or PAA can be achieved.

In general, the slight difference in the quality of MC64 and PAA is caused by the  $\varepsilon$ -scaling mechanism implemented in the PAA. It is dependent on the structure and values of the matrix whether the algorithm returns an optimal or near-optimal weight. Even in the sequential run of PAA, the time to compute the perfect matching can be tremendously reduced by allowing a small tolerance in the quality (see, e.g., *Hamrle3* or *av41092*). Note that the weights of *human\_gene1* and *mouse\_gene* are 0 because the product of the weighted edges in each matching is 1, and the logarithmic value is computed here.

A subset of the collection is taken to plot the speed improvement of PAA against MC64 using at most 1,024 Cray XE6 cores (see Fig. 10.13). In most cases, the sequential execution of PAA is slightly slower than MC64 due to the MPI-related overhead in the implementation. In Figs. 10.13(a) and 10.13(b), PAA needs a small number of iterations to find the perfect matching and achieves a maximum speed improvement of  $70\times$  against MC64. The set of matrices in Fig. 10.13(c) tends to be much harder for the PAA. Here, the price war scenario dominates the scalability behavior and limits the maximum speedup to  $7.8\times$ .

In the last group, in Fig. 10.13(d), two experiments are visualized: scalability for the hard instance *av41092*, and the stability of the performance scaling on exemplary matrices, *matrix\_9* and *ibm\_matrix\_2*, and their transposes. The scaling behavior of matrix *av41092* can be explained by the hybrid MPI–OpenMP implementation of the parallel algorithm. Running only on one compute node and increasing the number of used OpenMP threads improves the performance slightly. As soon as a second MPI process is involved in the computation, the performance drops due to the strongly connected components in the matrix. Unfortunately, it results in a large number of auction iterations. For matrices like *Hamrle3* or *av41092*, where a large number of auction iterations is needed, it is hard to achieve any reasonable speedup by a distributed matching algorithm [8]. However, a large performance gain against MC64 can be attained by run-



**Figure 10.13:** Speed improvement of the PAA over MC64 using up to 1,024 Cray XE6 cores shown in a log-log plot.

ning PAA on a single core. Comparing the performance scaling of the matrices and their transposes implies that the scaling behavior of the PAA may change drastically with the reordering of the matrix. In this case, the transposes of the matrices almost double the performance scaling.

### 10.3.2 Artificial Dense Bipartite Graphs

In order to get valuable insights into the behavior of the PAA for applications dealing with dense graphs, fully dense graphs with random edge weights were generated to conduct weak and strong scalability studies.

In a sequential experiment (see Table 10.9), the rounded weights of the matching and the timings are presented for the greedy heuristic, approximation algorithm, augmenting path implementation MC64, and PAUL. The greedy heuristic and the approximation algorithm return, for fully dense graphs, a maximum matching with near-optimal weight, but the approximation algorithm is more than one order of magnitude faster than the greedy approach. The time to compute a matching with  $\epsilon$ -PAA is

**Table 10.9:** Comparison of the quality of the matching between the greedy, MC64, the PAA, and  $\epsilon$ -PAA for dense graphs.

Problem	$W_1$	Greedy	Approx	MC64	PAA (#it)	$\epsilon$ -PAA (#it)
<i>rand10240</i>	10,238	99.93% 60.71 s	99.93% 4.22 s	100% 31.15 s	100% (19,961) 14.76 s	99.99% (4,617) 8.87 s
<i>rand15360</i>	15,358	99.95% 143.54 s	99.95% 9.50 s	100% 70.19 s	100% (10,681) 33.92 s	99.99% (22,554) 20.99 s
<i>rand20480</i>	20,478	99.96% 277.20 s	99.97% 16.85 s	100% 156.87 s	100% (61,021) 66.12 s	99.99% (7,271) 35.22 s
<i>rand28160</i>	28,158	99.97% 397.28 s	99.97% 31.87 s	100% 441.83 s	100% (53,576) 137.46 s	99.99% (7,285) 69.93 s

twice as much as the time of the approximation algorithm, but  $\epsilon$ -PAA finds a better quality of the matching. The matching quality of MC64 and PAA are in the same range, but the PAA is at most three times faster than the augmenting path implementation.

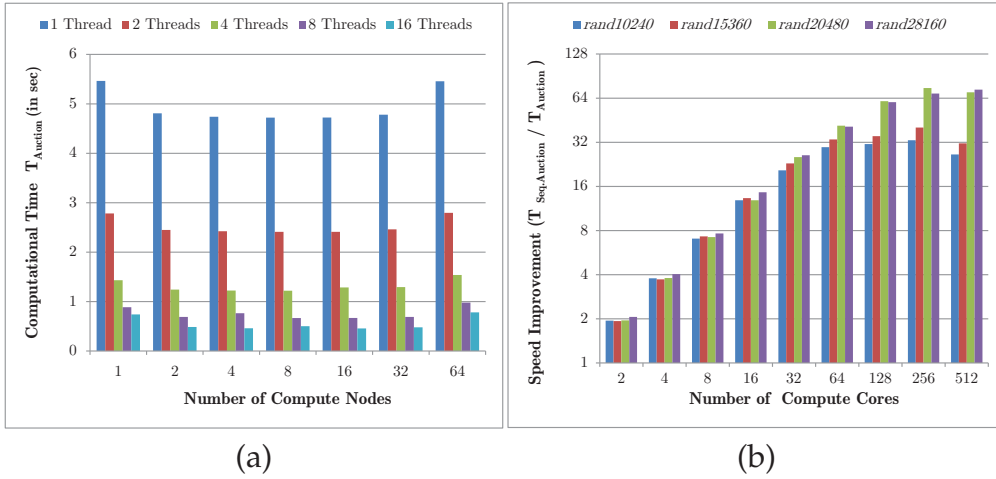
Weak and strong scalability results are presented in Fig. 10.14. In the weak scalability scenario in Fig. 10.14(a), the input size increases proportionally to the number of cores employed. Each compute node owns a subgraph and finds a maximum matching on the local part. For instance, an initial graph of size  $8,192 \times 8,192$  is generated for one node and the number of nodes is increased up to 64 (each using up to 16 threads, and 1,024 cores in total). As the number of columns and cores are doubled, the number of rows is halved. The resulting largest graph on 64 nodes has more than 4 billion edges that will be solved using up to 1,024 cores. Ideally, the run time is expected to remain constant. For the dense graphs the auction-based method shows an excellent scaling behavior.

For strong scalability in Fig. 10.14(b), dense matrices with random edge weights were generated which can still be stored on a single compute node; the largest one has more than 792 million weighted edges and is solved by the  $\epsilon$ -PAA using up to 512 compute cores.

### 10.3.3 Image Feature Matching

Graph matching [55, 152] plays an essential role in image-based 3-D reconstruction. Given a set of images that capture a scene from different perspectives, the goal is to partly infer the 3-D scene geometry.



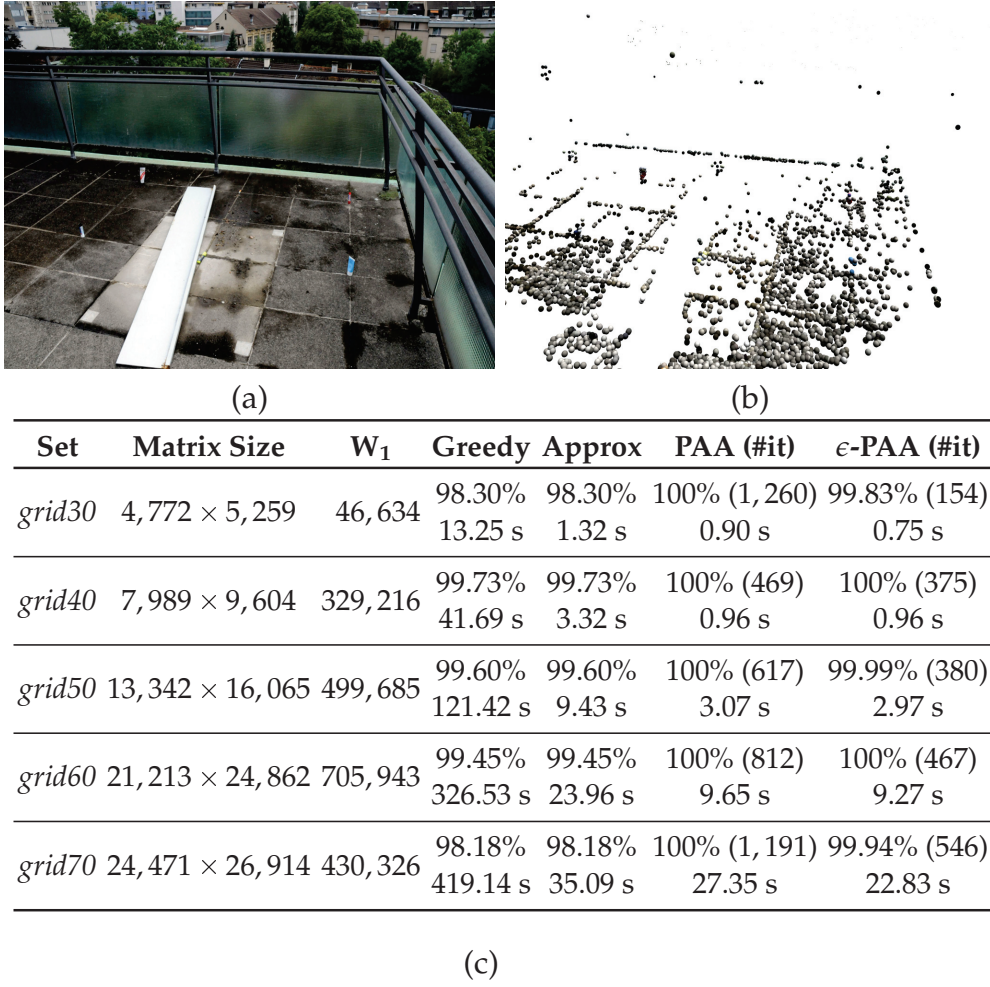


**Figure 10.14:** Weak (a) and strong (b) scalability results for the  $\epsilon$ -PAA in artificial dense graphs.

A common approach is the scale-invariant feature transform (SIFT) algorithm to detect features in images. So-called SIFT features, which densely cover a complete image over the full range of scales and locations, are extracted. Each feature consists of a 128-dimensional feature vector. The SIFT features are bundled to a feature set, and each feature set represents an image which is stored in a database. Now, given a collection of feature sets, the task is to identify similar features in all the images, resulting in an artificial representation of the image set. In Fig. 10.15(a), one image of nine photo perspectives is visualized. In Fig. 10.15(b), the structure of the original image can be reconstructed as a point cloud. If the model of the scene is appropriate enough, a typical application is to categorize a new image of the same or different scene into groups of similar images. The matching identifies similar features in a set of images [152].

A bipartite graph  $\mathcal{G}_b = (\mathcal{V}_1, \mathcal{V}_2, \mathcal{E}, w)$  is constructed by two images as follows: each feature of the first image becomes a vertex in  $\mathcal{V}_1$ , and each feature of the second image a vertex in  $\mathcal{V}_2$ . The weight of an edge corresponds to the Euclidean distance between two features of different images. The matching problem can be posed as finding as many correspondences as possible between the two images with the objective being to minimize the overall distance, i.e.,  $\min \sum_{(i,j) \in M} w_{ij}$ . Typically,  $\mathcal{G}_b$  is unbalanced due to a different number of features per image.

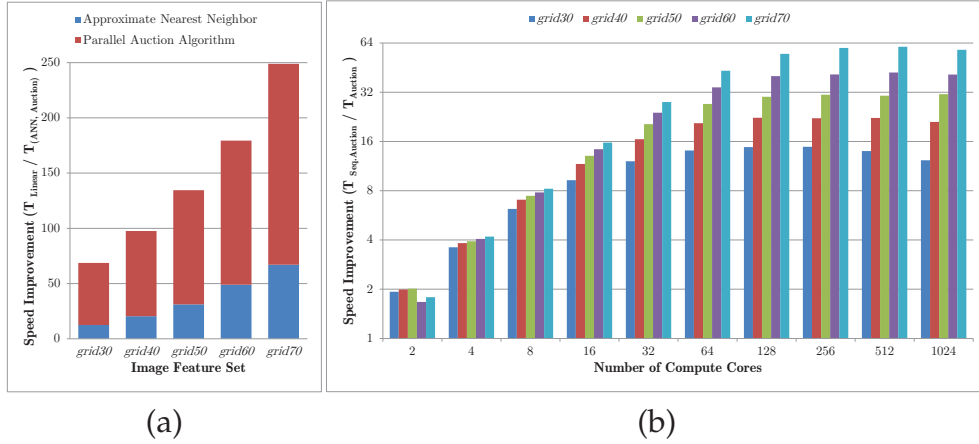
Since the auction algorithm is designed for matching problems which



**Figure 10.15:** (a) One of the nine photo perspectives of a narrow-area scene, called “grid.” (b) Point cloud visualization from the estimated camera perspective constructed after matching of extracted features of the images. (c) Comparison of the quality of the matching between the greedy and approximation algorithms, PAA, and  $\epsilon$ -PAA when solving one instance per feature set.

maximize the objective function, the edge weights are further scaled: the edge weight is replaced by the quotient of the overall maximum weight and the edge weight.

Linear search between all feature sets is only viable for small sets and lower-dimensional features. For higher-dimensional vector spaces, approximation algorithms have been developed, such as the approximate nearest neighbor (ANN) search [12]. While using an input set with a high number of features is computationally more expensive, it allows a more detailed and more accurate reconstruction. This is the main reason



**Figure 10.16:** (a) Comparison of the ANN search, linear search, and the PAA for the feature set grid. (b) Log-log plot of the speed improvement of parallel auction on the Cray XE6 against its sequential run.

for constructing the bipartite graph and for using maximum weighted matching algorithms to identify feature correspondences among the images. Once the feature sets have been matched, some feature correspondences are discarded due to noise in the data. Given that enough corresponding features have been found, the scene geometry can be inferred successfully [221].

In order to benchmark the performance scaling of the PAA, images of a scene with different resolutions are generated where the images differ in the number of features extracted from them. The feature sets are called *grid30*, *grid40*, ..., *grid70*, where the sets differ in the number of features, ranging from 3,189 up to 31,616. In a sequential experiment, the qualities of PAUL, the greedy algorithm, and the approximation algorithm are compared to each other as listed in the table of Fig. 10.15(c). The  $\epsilon$ -PAA outperforms the greedy algorithm as well as the approximation algorithm regarding timings and quality. Each feature set contains nine images and every image is compared to the other images in the same feature set. The average time is measured over all matching calls and represents the final run time for each image set.

For large unbalanced bipartite graphs, the PAA scales almost linearly up to 64 compute cores, and speedup is achieved up to 512 compute cores (see Fig. 10.16(b)). There is a strong relationship between the number of features per image in the feature set and the maximum speedup: the larger the image sizes, the larger the speed improvement. However, the

**Table 10.10:** *Biological data for the global alignment of protein-protein interaction networks.*

Species	Biological Term	Dataset	$ \mathcal{V} $	$ \mathcal{E} $
fruit fly	<i>Drosophila Melanogaster</i>	<i>dmela</i>	7,518	25,830
human	<i>Homo Sapiens</i>	<i>hsapi</i>	9,633	36,386
yeast	<i>Saccharomyces Cerevisiae</i>	<i>scere</i>	5,499	31,898

communication of the prices limits the speedup when using more than 256 computing cores.

In Fig. 10.16(a), the computational timings of the parallel auction are compared to existing solution approaches. The PAA outperforms the sequential ANN approach by a factor of three.

It has been found visually that the results of using existing algorithms and the auction-based implementation are nearly equivalent. However, in some cases in which the number of corresponding features was rather low, the PAA was able to estimate more perspectives of the input set precisely than the existing approaches.

### 10.3.4 Graph Similarity

Identifying common connected subgraphs in two graphs of the form  $\mathcal{G} = (\mathcal{V}, \mathcal{E}, w)$  is a well-known problem in computational science and engineering. In this section, small-sized protein-protein interaction (PPI) networks, and large-sized web graphs and social networks are analyzed to demonstrate the parallel scalability of the PAUL in a parallel graph similarity framework.

### Protein-Protein Interaction Networks

In the global alignment of PPI networks, graph-theoretic commonalities and evolutionary relations between the graphs are observed, where physically interacting proteins are represented as edge-connected vertices [135, 171, 219]. Identifying topological regions of similarity between graphs of different species reveals insights into the functional organization and coherence of subnetworks. Specifically, if connected subgraphs are conserved across species, they likely correspond to shared functions across and within subgraphs. This can be used for annotating proteins (by mapping annotations across species), inferring missing interactions,

**Table 10.11:** Comparison of the matching quality between the greedy and the parallel auction implementations.

PPI pair	$W_1$	Greedy	Approx	PAA (#it)	$\epsilon$ -PAA (#it)
<i>dmela-scere</i>	0.0407	98.69% 20.88 s	98.72% 3.87 s	100% (4,202) 14.76 s	99.71% (126) 1.46 s
<i>dmela-hsapi</i>	0.0265	98.51% 37.44 s	98.40% 5.91 s	100% (1,668) 5.85 s	99.88% (164) 1.86 s
<i>hsapi-scere</i>	0.0249	99.10% 27.16 s	99.01% 6.05 s	100% (1,510) 4.98 s	99.84% (144) 1.72 s

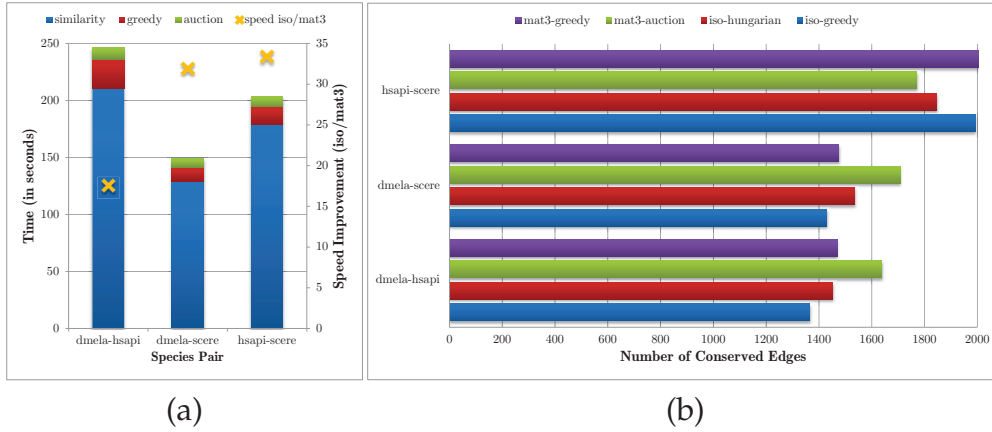
and drawing functional orthologies. Three publicly available PPI networks with reliable data are considered for the benchmarks as presented in Table 10.10.

In order to find the topological similarity between two species, a two-step approach has been developed: the first step is to construct the similarity matrix, which contains the information about similarity score between vertices. The second step extracts matching pairs of vertices with a high similarity score (see Section 9.1). In this step, the bipartite graph matching problem is a computational bottleneck: it takes hours if not days applying an exact matching method like the widespread Hungarian method to the similarity matrix [171, 219]. In contrast, PAUL is a fast and efficient approach.

In this section, it will be shown that auction algorithms can be used to quickly extract similar protein-protein pairs with a quality comparable to existing approaches. Therefore, the auction algorithms are executed sequentially on a single core.

In Table 10.11, the greedy heuristic and the approximation algorithm are compared to the two variants in PAUL. The weight of the matching is computed by  $W_1 = \sum_{(i,j) \in \mathcal{M}} |w_{ij}|$ . In the columns of the different algorithms, the achieved weight of the matching algorithm is given in relation to  $W_1$  and the total time to compute the matching. In the case of the PAAs, additionally, the number of iterations is presented. It can be concluded that the adaptive  $\epsilon$ -PAA is the best choice among the considered algorithms regarding quality and time. The weight of the solution is nearly as good as the optimal one, while only one tenth of the number of iterations is needed.

These experiments have encouraged embedding the approximative

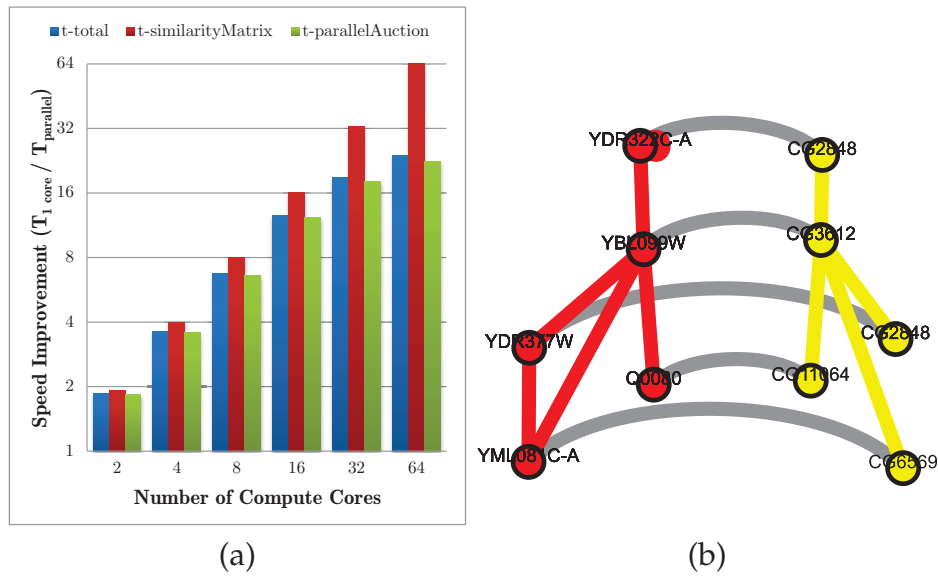


**Figure 10.17:** Time to compute the global alignment of three species pairs and speedup against IsoRank (a). Number of conserved edges using different matching routines (b).

auction algorithm and a greedy algorithm into a similarity framework, *mat3*, where the similarity matrix is constructed as described in [101]. The approach is then compared to a state-of-the-art global alignment tool, *IsoRank* (iso) [219], which implements the same similarity matrix construction and applies the Hungarian method and a greedy matching implementation to the matrix. *IsoRank* is available online, and the tool provides the time required for the full similarity computation including the similarity construction and the subsequent graph matching algorithms. Hence, total timings — including both the similarity matrix construction and the computation of matching pairs — of both frameworks are compared to each other.

The time required for the similarity computation, for the greedy algorithm, and for the auction algorithm (in seconds) for the PPI pairs, are plotted along with the speed improvement over *IsoRank* in Fig. 10.17(a). Approximately 4 minutes are sufficient to compute the global alignment of PPI network pairs where — mainly contributed by the auction implementation — *mat3* is sequentially more than one order of magnitude faster than *IsoRank*. As the computation can be accelerated by *mat3*, the quality of the matching pairs must be analyzed. As a quality indicator, the number of conserved edges in the alignment graph is reported in Fig. 10.17(b). The attained numbers for *mat3* are comparable or even better than the *IsoRank* numbers for the PPI pairs.

When using a parallel similarity framework, consisting of the NSD approach (see Algorithm 9.1) and PAUL, to compute the similarity be-



**Figure 10.18:** (a) Scalability results for the PPI pair *dmela-scere*. (b) Zoom into a common connected subgraph of the protein-protein pair *dmela-scere* with specific functional coherence, i.e., RNA splicing.

tween PPI networks, a dramatic time reduction is gained from parallelization on the Cray XE6 as shown in Fig. 10.18(a).

The scalability of the total similarity computation time, “t-total,” the time to compute the similarity matrix, “t-similarityMatrix,” and the time of the  $\epsilon$ -PAA, “t-parallelAuction,” are shown. The x-axis displays the number of compute cores, and the y-axis shows the speed improvement of the framework across its sequential execution. Roughly 3 seconds are enough to extract the matching pairs for the protein-protein pair *dmela-scere* using 64 cores. In contrast, using sequential state-of-the-art approaches like IsoRank [219], about 1.5 hours were required for the total process to obtain a comparable quality of the solution. It is also possible to find specific functional coherence between the common connected subgraphs of the two proteins as shown in Fig. 10.18(b).

It can be summarized that the similarity of protein-protein interaction networks can be computed with one to two order magnitude speedup against the existing IsoRank approach when using auction algorithms while yielding a comparable topological and biological quality.



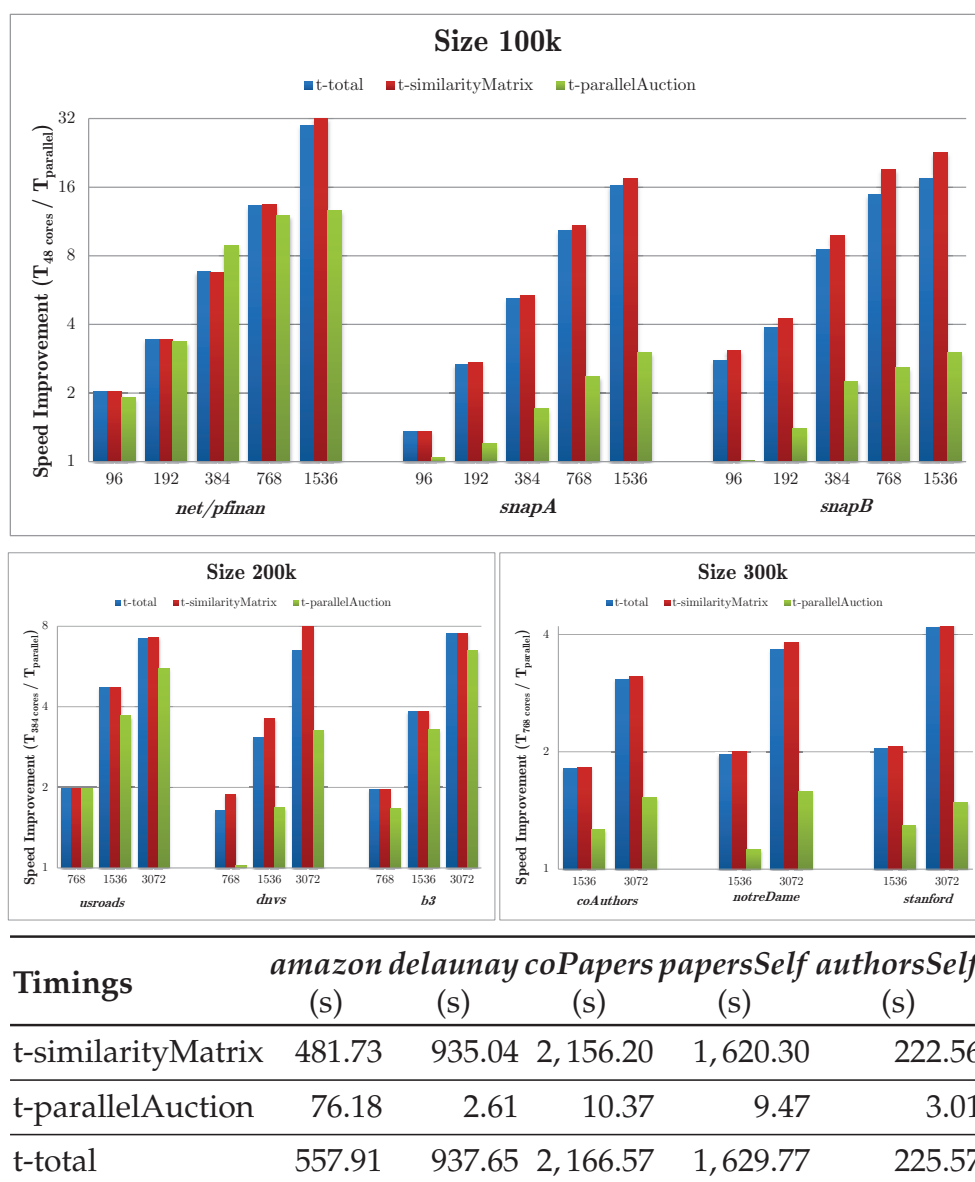
**Table 10.12:** *Characteristics of large web and real life graphs.*

Size	Pair	Graph	#Vertices	#Edges
100k	<i>net/pfinan</i>	<i>net4-1</i>	88,343	1,265,035
		<i>pfinan512</i>	74,752	335,872
	<i>snapA</i>	<i>soc-slashdot090221</i>	82,144	549,202
		<i>soc-slashdot090216</i>	81,871	545,671
	<i>snapB</i>	<i>soc-slashdot0902</i>	82,168	948,464
		<i>soc-slashdot0811</i>	77,360	905,468
200k	<i>usroads</i>	<i>usroads</i>	129,164	165,435
		<i>usroads-48</i>	126,146	161,950
	<i>dnvs</i>	<i>halfb</i>	224,617	6,306,219
		<i>fullb</i>	199,187	5,953,632
	<i>b3</i>	<i>m133-b3</i>	200,200	800,800
		<i>shar_te2-b3</i>	200,200	800,800
300k	<i>coAuthors</i>	<i>coAuthorsDBLP</i>	299,067	977,676
		<i>coAuthorsCiteseer</i>	227,320	814,134
	<i>notreDame</i>	<i>NotreDame_www</i>	325,729	929,849
		<i>web-NotreDame</i>	325,729	1,497,134
	<i>stanford</i>	<i>Stanford</i>	281,903	2,312,497
		<i>web-Stanford</i>	281,903	2,312,497
>300k	<i>amazon</i>	<i>amazon0505</i>	410,236	3,356,824
		<i>amazon0601</i>	403,394	3,387,388
	<i>delaunay</i>	<i>delaunay_n19</i>	524,288	1,572,823
		<i>delaunay_n18</i>	262,144	786,396
	<i>authorsSelf</i>	<i>coAuthorsCiteseer</i>	227,320	814,134
		<i>coAuthorsCiteseer</i>	227,320	814,134
	<i>coPapers</i>	<i>coPapersDBLP</i>	540,486	15,245,729
		<i>coPapersCiteseer</i>	434,102	16,036,720
	<i>papersSelf</i>	<i>coPapersCiteseer</i>	434,102	16,036,720
		<i>coPapersCiteseer</i>	434,102	16,036,720

## Collection of Web and Real Life Graphs

Motivated by the success of analyzing small-sized biological networks, 36 moderate to large real life graphs from the *University of Florida Sparse Matrix Collection* [63] were examined to identify common connected sub-graphs [136]. The motivation behind the analysis of web graphs is, for instance, to discover evolutionary changes of web graphs over time and to detect anomalies between two graphs with different time stamps [183].





**Figure 10.19:** Speed improvement and timing results (in secs) from the compute intensive steps in the graph similarity framework.

Table 10.12 lists the size of the category along with the internal name of the pair, as well as structural properties of each graph. Two graphs are grouped to a pair if the adjacency matrices have a similar pattern. Five groups are created depending on the matrix sizes from 100k up to 500k. For the pairs categorized into the group greater than 300k, all 3,072 compute cores are needed to be able to store the fully dense graphs. The similarity computation of large web graphs can be drastically accelerated

**Table 10.13:** *Quality measurements, number of conserved edges (CE) and the similarity rate (rate), for the analyzed pairs.*

Pair	Time (s)	#Cores	#CE	Rate
<i>net/pfinan</i>	796	48	74,778	0.22
<i>snapA</i>	2,688	48	14,296	0.02
<i>snapB</i>	1,497	48	77,617	0.09
<i>usroads</i>	281	384	2,666	0.02
<i>dnvs</i>	880	384	1,750,799	0.29
<i>b3</i>	1,593	384	29,217	0.15
<i>coAuthors</i>	659	768	85,437	0.11
<i>notreDame</i>	764	768	113,992	0.12
<i>stanford</i>	615	768	107,968	0.05
<i>amazon</i>	558	3,072	46,278	0.01
<i>deLaunay</i>	938	3,072	112,152	0.14
<i>authorsSelf</i>	226	3,072	814,134	1.00
<i>coPapers</i>	2,167	3,072	3,520,545	0.23
<i>papersSelf</i>	1,630	3,072	16,036,720	1.00

using a parallel distributed framework consisting of the embarrassingly parallel implementation of the NSD approach (see Algorithm 9.1) and PAUL. Thus, the parallel NSD implementation generates a distributed rectangular dense similarity matrix which is the input for PAUL in order to detect high quality matching pairs.

Strong scalability tests are conducted on the Cray XE6 to validate the scaling behavior of the framework and the following two crucial parts: the similarity matrix construction and the matching pair extraction.

In Fig. 10.19, the scalability of the parallel similarity framework is presented for the three groups 100k, 200k, and 300k using up to 3,072 cores. The focus is on the timings of the compute intensive parts: the scaling of the total time of the similarity computation framework, “t-total,” the time to compute the similarity matrix, “t-similarityMatrix,” and the time required for the  $\epsilon$ -PAA, “t-parallelAuction.” The x-axis provides the number of compute cores, and the y-axis indicates the speed improvement over the following number of compute cores: in the 100k case, 48 cores, in the 200k case, 384 cores, and in the 300k case, 768 cores were used. Almost linear speedup is reported for the embarrassingly parallel similarity matrix construction, and also for the overall time. The parallel auction

matching scales reasonably well, but takes up only a small fraction of the overall time.

In the table, which is included in Fig. 10.19, the timings (in seconds) are presented for the largest pairs  $> 300k$  when all available 3,072 compute cores are employed.

In Table 10.13, the quality measurements, number of conserved edges (#CE) in the alignment graph and the similarity rate (Rate), are computed. The similarity rate provides a relative number for the similarity of two graphs. The similarity rate of the pairs *net/pfinan*, *dnvs*, and *coPapers* is greater than 20% which indicates that the matrices in pairs have a similar pattern. It can be also employed to check the robustness of the framework if used in the context of self-similarity (matching a graph with itself): in the optimal case it is expected to obtain a number of conserved edges equal to the number of edges in the graph. This is indeed the case for the pairs *authorsSelf* and *papersSelf*. Additionally, the total time (in seconds) is provided for the similarity computation using the indicated number of compute cores.

It can be concluded that the parallel auction-based weighted matching implementation finds matchings efficiently in already distributed matrices, and similarity computations can be processed with matrices two orders of magnitude larger than currently possible.



## **Part V**

# **Conclusions & Outlook**



## Chapter 11

---

# Conclusions and Outlook

---

In this thesis, the crucial role of finding weighted graph matchings and subgraphs in computational science has been demonstrated by solving various data intensive applications.

In order to obtain weighted matchings in parallel, auction algorithms represent very attractive approaches to run on massively parallel architectures. A parallel distributed auction algorithm based on a hybrid MPI–OpenMP programming model is developed that can be drastically accelerated by introducing different  $\varepsilon$ -scaling strategies. Especially for attaining weighted matchings in dense bipartite graphs, an adaptive  $\varepsilon$ -scaling mechanism is proposed that achieves very good performance scalability on a Cray XE6. For instance, in the global alignment of protein-protein interaction networks, the time to compute weighted matchings is dropped from hours, using the Hungarian method, to seconds. In sparse linear algebra, the graphs are even more challenging for the distributed auction algorithm and, at least, a substantial acceleration compared to existing methods and implementations based on augmented paths has been shown. Motivated by the lack of existing software libraries for the matching problem, PAUL has been implemented as a stand-alone software library. The implementation allows a drastic acceleration of the matching process and is able to compute matchings in very large bipartite graphs with millions of vertices and billions of edges, which could not be matched efficiently before.

The importance of finding weighted subgraphs has been motivated by its need in the hybrid linear solver PSPIKE. Hence, general prerequisites for the performance of PSPIKE are reordering routines embed-

ded into its preprocessing phase, which are obtained through solving the graph partitioning, the bipartite graph matching, and, in particular, the weighted dense subgraph problem.

In this thesis it has been shown that reorderings obtained by finding weighted subgraphs have a significant impact on the convergence speed of PSPIKE; in the worst case, it happens that the solver does not converge when the weighted subgraph problem is not appropriately solved. Two greedy heuristics and a (1+1)-EA were employed. Before applying these heuristics to the graph representation of the matrix, quality measures for determining vertex weights are introduced which indicate the overall contribution of the vertex to the subgraph. It can be concluded from the benchmark results that the (1+1)-EA computes the most efficient solutions for the problem with regard to both quality and time.

PSPIKE enters the new era of hybrid sparse linear equation system solvers, which will gain importance due to the fact that optimization and simulation in many data intensive applications in computational science solvers rely on linear equation system solvers, which require up to 99% of the entire solving time; examples include a fluid solver for arterial flow simulations or an interior point optimizer for PDE-constrained optimization. Thus, the acceleration of linear equation system solvers is a worthy endeavor and the computational power of supercomputers with thousands of compute cores offers a great opportunity to develop parallel linear solvers.

Current alternatives to solving linear equation systems in parallel are, on the one hand, problem-specific preconditioned iterative solvers, which are crafted to match a very specific problem structure and sensitive to even small alterations in the problem structure; on the other hand, there are direct linear solvers, which suffer from well-known limitations in performance scalability. PSPIKE combines the robustness of direct linear solvers with the performance scalability of iterative solvers, and scales well up to thousands of compute cores shown on a Cray XK6 architecture. Additionally, it offers the flexibility to solve linear equation systems accurately on a variety of problem classes. Thus, PSPIKE can be viewed as a *general purpose solver* or *black-box solver* as it does not require any knowledge of the type of the problem. The purpose of the hybrid solver PSPIKE is that the solving of many data intensive applications will benefit from the performance of PSPIKE, and the solver will hopefully support researchers and practitioners to solve problems so large that they could not have been solved using conventional parallel linear solvers.



PSPIKE runs on massively parallel hardware architectures and follows the domain decomposition approach. The performance of PSPIKE relies heavily on the quality of its preconditioner, which is constituted of heavy-weighted diagonal block and coupling block matrices, that are extracted from the original matrix. The diagonal and coupling block matrices should contain most of the entries of the original matrix in order to let the hybrid solver converge fast. The construction of this PSPIKE structure inherently depends on the quality of graph algorithms.

PSPIKE represents only one instance of parallel software libraries and toolkits in which graph algorithms are crucial. Although the parallelization of distributed graph algorithms is considered a challenging task due to irregular memory access patterns and low arithmetic intensity, some progress has been made in the development of efficient parallel software libraries for graph problems.

## Future Work and Directions

Concerning the parallel auction algorithm library, PAUL, and the finding of subgraphs further research needs to be done in several directions.

The  $\varepsilon$ -scaling strategy has a high influence on the convergence and the matching quality of the algorithm. In order to increase robustness of the algorithm, different  $\varepsilon$ -scaling strategies must be implemented for solving dense and sparse bipartite matching problems and a tool exploiting the techniques of machine learning should “automatically” adapt the “best” strategy to the input data.

As software in parallel graph matching is scarce, in addition to the auction algorithm implementation, PAUL, parallel approximation algorithms and maximum cardinality matching algorithms should be implemented in a single “matching toolbox.” This would allow researchers to easily compare the effect of a particular matching algorithm in their applications to other algorithms. Promising candidates for parallelization of a maximum weighted matching algorithm are the use of linear programming techniques (e.g., the simplex method or interior point methods), or finding matchings via shortest path computations [41].

In this thesis, only graph matchings on bipartite graphs have been studied. The question is still open if auction algorithms can be used for matching general graphs which is of considerable interest in current multilevel graph algorithm frameworks, such as multilevel graph parti-

tioner [65, 125, 229]; the challenge consists in dealing with the fact that a vertex in a general graph represents a buyer and an object, simultaneously.

An issue in the auction algorithm which needs to be definitely addressed in the future is the long tail problem [8], i.e., in the first iterations most of the vertices are matched, but then the price war scenario dominates the progress in the algorithm. Thus, only a small number of unmatched buyers compete for the remaining free objects which results in a large number of auction iterations. For instance, the long tail problem of PAUL can be seen when solving the matching problem for the matrices *av41092* or *Hamrle3* (see Sec. 10.3).

For the problem to find weighted subgraphs it would be interesting to compare the  $(1 + 1)$ -EA with population-based metaheuristics like ant colony or particle swarm optimizer. Currently, the  $(1 + 1)$ -EA is only tested on applications embedded into the numerical hybrid linear solver PSPIKE. However, other large-scale data intensive applications like the community detection in social networks and the finding of important substructures in biological networks are challenging tasks.

Most data intensive applications, for instance, of social networks and computational biology, generate a huge amount of heterogeneous data. In order to represent, the all complex networks, vertices and edges will not be labeled with a single numerical value, but with metadata describing their properties and complex structures [20]. Thus, common sparse storage formats like compressed sparse row or coordinate formats will not accurately model the data and, consequently, graph algorithms for finding matchings or subgraphs will suffer from the lack of modeling. Hence, new efficient ontologies and data models are required to provide a sufficient modeling for the complex data. However, as a consequence of new ontologies, graph algorithms need to be reengineered and adapted to deal with the metadata, which have an impact on the computational complexity of original graph algorithms like auction-based matching algorithms.

A property of many data intensive applications is that data evolve over time as vertices and edges can be deleted from and added to the corresponding graph, and metadata can change over time. However, the auction algorithm is only able to compute matchings for a static graph. The question arises of how to design and implement a parallel auction algorithm which is able to deal with these “dynamic” graphs.

It is often required to compute matchings of one graph to all other

graphs in a database. In image feature matching, images of all sizes are typically stored in a database and need to be compared to a single original image in order to classify the image into a similar image collection set. Similarly in bioinformatics, the multiple alignment of sequences is of considerable interest which leads to a one-to-many graph matching problem [189]. In the auction algorithm, only one-to-one graph matching problems can be currently solved, but it would be interesting to benchmark an auction algorithm with the ability to compute one-to-many graph matchings against state-of-the-art image matching and sequencing alignment methods.

Although a great attention is paid to approaching the exascale era in software and hardware [15], optimizing parallel graph algorithms is still challenging on supercomputers in the terascale era. Modern architectures feature a fast memory bandwidth, but typically a high latency, which result in poor performance due to the irregular memory access pattern and the pure data locality of graph algorithms: on shared memory architectures, processors are competing for the same single memory subsystem which drops the performance of the algorithm. On distributed memory architectures, the distribution of the graph plays a dominant role for the data locality on each compute node and a key aspect is the integration of communication avoiding and reducing mechanisms into the graph algorithm.

It is an open question if accelerators like GPUs will enhance the performance of graph algorithms, but massively multithreaded architectures like the Cray XMT seem to be more promising for graph algorithms [3, 138, 145]. These shared memory architectures stress a large globally memory to store massive data and feature processors which control many light-weighted threads running on so-called “streams.” On distributed memory architectures, the Intel Many Integrated Core (MIC) architecture seems to be an interesting multicore to obtain better performance for graph algorithms on the Intel MIC clusters [207]. Thus, designing, benchmarking, and comparing graph algorithms, in particular, matching algorithms, across different shared memory and distributed memory architectures will be a further task which introduces the potential to be a valuable standard benchmark for the Graph500 [223]. Visualizing the huge graphs and highlighting the graph matching set in order to better analyze and understand the output of the algorithm would also an interesting effort on these multicore clusters [231].

There are several opportunities for future research regarding the im-

provement of the hybrid linear solver PSPIKE.

In PSPIKE, the direct linear solver PARDISO as well as the iterative linear solver BICGSTAB are employed. Different interfaces to other parallel direct solvers including WSMP [108] and SUPERLU [150] would increase the flexibility of the hybrid solver. In particular, parallel replacements of BICGSTAB with other iterative solvers (e.g., with GMRES [199]) can make a significant difference for the performance of the entire solver since the convergence rate of iterative solvers is dependent on the application data, their numerical properties, and the strength of the preconditioner within the iterative solver.

Second, problem-specific “automatic parameter adaptation” of the inputs, bandwidth and inner tolerance level, should be addressed. The notion “automatic” refers to the process of finding the optimal trade-off between a small bandwidth and high coverage of elements, ideally around 99%, for all subdomains automatically. A small bandwidth reduces the size of the linear system in the inner solving step, and a high coverage strengthens the preconditioner and improves the convergence rate of the entire solver.

Although, even if a good coverage of the elements has been achieved, there might be an issue that the reduced spike system (see Eq. 6.5) is not solved accurately to the given desired inner tolerance with the preconditioned BICGSTAB solver which could be also observed when solving the Helmholtz equations [37]. The size of the system is  $2k(P - 1)$  where  $k$  is the bandwidth and  $P$  the number of subdomains. Each row in the system contains  $2k$  entries, while the rows of the first and last subdomain have  $k$  entries, respectively. As the systems need to be solved twice per outer iteration, alternative parallel direct or iterative linear solvers must solve these linear systems efficiently. Currently, the “on-the-fly” variant of the SPIKE algorithm is the building block in PSPIKE [191]. However, including into PSPIKE the “recursive” and “direct” schemes in combination with the adaptive strategies implemented in [176] would enhance the robustness and flexibility of the solver.

As there are a few other hybrid solvers, a comparison between them in various real life applications would be interesting. It might be possible to classify the real life applications into categories which determine whether PSPIKE will perform well.

There are open questions in PSPIKE that need to be addressed in the future: The first question is if it is possible to obtain the same number of outer BICGSTAB iterations and the same solution of the linear system

while increasing the number of domains and still gaining speedup? This will be especially interesting when PSPIKE runs on a large number of compute cores. A further question is if it is possible to develop a graph partitioner that is able to create the “optimal” PSPIKE structure for the problem? Currently, the problem has been split into several steps. And the third question is if there is a way to analyze the convergence rate and performance of PSPIKE for general linear systems theoretically?



---

# Bibliography

---

- [1] DIMACS Implementation Challenge 10th. Graph partitioning and graph clustering. <http://www.cc.gatech.edu/dimacs10/>. Accessed October 2011. [cited at p. 25]
- [2] Réka A. and A.-L. Barabási. Statistical mechanics of complex networks. *Review of Modern Physics*, 74:47–97, 2002. [cited at p. 1]
- [3] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader. Scalable graph exploration on multicore processors. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1–11. IEEE, 2010. [cited at p. 153]
- [4] T. Aittokallio and B. Schwikowski. Graph-based methods for analysing networks in cell biology. *Briefings in bioinformatics*, 7(3):243–255, 2006. [cited at p. 18]
- [5] E. Alba. *Parallel metaheuristics: a new class of algorithms*. Wiley-Interscience, 2005. [cited at p. 28, 30]
- [6] P. R. Amestoy, I. S. Duff, and J. Y. L’Excellent. Multifrontal parallel distributed symmetric and unsymmetric solvers. *Computer Methods in Applied Mechanics and Engineering*, 184(2–4):501–520, 2000. [cited at p. 8, 62]
- [7] O. Amini, S. Pérennes, and I. Sau. Hardness and approximation of traffic grooming. *Theoretical Computer Science*, 410(38–40):3751–3760, 2009. [cited at p. 29]
- [8] C. Anderson. *The long tail: Why the future of business is selling less of more*. Hyperion Books, 2008. [cited at p. 132, 152]
- [9] R. Aringhieri and R. Cordone. Comparing local search metaheuristics for the maximum diversity problem. *Journal of Operational Research Society*, 62(2):266–280, 2011. [cited at p. 29]

- [10] S. Arora, D. Karger, and M. Karpinski. Polynomial time approximation schemes for dense instances of NP-hard problems. In *Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*, pages 284–293. ACM, 1995. [cited at p. 27]
- [11] P. J. Artymiuk, A. R. Poirrette, H. M. Grindley, D. W. Rice, and P. Willett. A graph-theoretic approach to the identification of three-dimensional patterns of amino acid side-chains in protein structures. *Journal of Molecular Biology*, 243(2):327–344, 1994. [cited at p. 17]
- [12] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *Journal of the ACM (JACM)*, 45(6):891–923, 1998. [cited at p. 136]
- [13] Y. Asahiro, K. Iwama, H. Tamaki, and T. Tokuyama. Greedily finding a dense subgraph. *Journal of Algorithms*, 34(2):203–221, 2000. [cited at p. 29]
- [14] M. Ashburner, C. A. Ball, J. A. Blake, D. Botstein, H. Butler, J. M. Cherry, A. P. Davis, K. Dolinski, S. S. Dwight, J. T. Eppig, et al. Gene ontology: tool for the unification of biology. *Nature genetics*, 25(1):25, 2000. [cited at p. 101]
- [15] S. Ashby, P. Beckman, J. Chen, P. Colella, B. Collins, D. Crawford, J. Dongarra, D. Kothe, R. Lusk, P. Messina, et al. The opportunities and challenges of exascale computing. *Summary Report of the Advanced Scientific Computing Advisory Committee (ASCAC) Subcommittee, Fall*, 2010. [cited at p. 153]
- [16] C. Aykanat, B. B. Cambazoglu, and B. Uçar. Multi-level direct  $K$ -way hypergraph partitioning with multiple constraints and fixed vertices. *Journal of Parallel and Distributed Computing*, 68:609–625, 2008. [cited at p. 18]
- [17] A. Azad, M. Halappanavar, S. Rajamanickam, E. G. Boman, A. Khan, and A. Pothen. Multithreaded algorithms for maximum matching in bipartite graphs. In *2012 International Symposium on Parallel and Distributed Processing Symposium*, pages xxxx–xxxx. IEEE Press, 2012. [cited at p. 20]
- [18] D. A. Bader and K. Madduri. SNAP, small-world network analysis and partitioning: An open-source parallel graph framework for the exploration of large-scale networks. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–12, 2008. [cited at p. 2]
- [19] S. Balay, J. Brown, K. Buschelman, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. PETSc web page. <http://www.mcs.anl.gov/petsc>. Accessed January 2012. [cited at p. 114]



- [20] D. Barabási. Barabasi lab. <http://www.barabasilab.com/>. Accessed April 2012. [cited at p. 152]
- [21] S. T. Barnard, A. Pothén, and H. D. Simon. A spectral algorithm for envelope reduction of sparse matrices. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, Supercomputing '93, pages 493–502, New York, NY, USA, 1993. ACM. [cited at p. 27]
- [22] M. Bayati, M. Gerritsen, D. F. Gleich, A. Saberi, and Y. Wang. Algorithms for large, sparse network alignment problems. In *2009 Ninth IEEE International Conference on Data Mining*, pages 705–710. IEEE, 2009. [cited at p. 96]
- [23] S. Belongie, J. Malik, and J. Puzicha. Shape matching and object recognition using shape contexts. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24:509–522, 2001. [cited at p. 18]
- [24] G. Benedetti and S. Morosetti. A graph-topological approach to recognition of pattern and similarity in RNA secondary structures. *Biophysical chemistry*, 59(1):179–184, 1996. [cited at p. 17]
- [25] M. Benzi. Preconditioning techniques for large linear systems: a survey. *Journal of Computational Physics*, 182:418–477, November 2002. [cited at p. 62]
- [26] M. W. Berry and A. Sameh. Multiprocessor schemes for solving block tridiagonal linear systems. *The International Journal of Supercomputer Applications*, 1(3):37–57, 1988. [cited at p. 63]
- [27] D. P. Bertsekas. *Linear network optimization: algorithms and codes*. MIT Press, Cambridge, MA, USA, 1991. [cited at p. 51, 52]
- [28] D. P. Bertsekas and D. A. Castañón. Parallel synchronous and asynchronous implementations of the auction algorithm. *Parallel Computing*, 17(707–732), 1991. [cited at p. 19, 38, 53]
- [29] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and distributed computation: numerical methods*. Prentice-Hall, Inc., 1989. [cited at p. 35, 53]
- [30] R. Bisseling. Mondriaan for sparse matrix partitioning. <http://www.staff.science.uu.nl/~bisse101/Mondriaan/>. Accessed January 2012. [cited at p. 90]
- [31] R. H. Bisseling. *Parallel scientific computation – a structured approach using BSP and MPI*. Oxford University Press, 2004. [cited at p. 2, 44]
- [32] R. H. Bisseling, B. O. Fagginger Auer, A. N. Yzelman, T. van Leeuwen, and U. V. Çatalyürek. Two-dimensional approaches to sparse matrix partitioning. In U. Naumann and O. Schenk, editors, *Combinatorial Scientific*

- Computing*, Chapman-Hall/CRC Computational Science. Taylor & Francis, 2011. [cited at p. 23, 25]
- [33] BLAS. Basic linear algebra subprograms. <http://netlib.org/blas/>. Accessed November 2011. [cited at p. 67, 88]
  - [34] BLAST. Basic local alignment search tool. <http://blast.ncbi.nlm.nih.gov/Blast.cgi>. Accessed November 2011. [cited at p. 98]
  - [35] V. D. Blondel, A. Gajardo, M. Heymans, P. Senellart, and P. Van Dooren. A measure of similarity between graph vertices: Applications to synonym extraction and web searching. *SIAM Review*, 46(4):647–666, 2004. [cited at p. 95, 96]
  - [36] C. Blum and A. Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys (CSUR)*, 35(3):268–308, 2003. [cited at p. 27]
  - [37] M. Bollhöfer, M. J. Grote, and O. Schenk. Algebraic multilevel preconditioner for the Helmholtz equation in heterogeneous media. *SIAM Journal on Scientific Computing*, 31(5):3781–3805, 2009. [cited at p. 61, 154]
  - [38] E. B. Boman, U. V. Çatalyürek, C. Chevalier, and K. D. Devine. Parallel partitioning, ordering, and coloring in scientific computing. In U. Naumann and O. Schenk, editors, *Combinatorial Scientific Computing*, Chapman-Hall/CRC Computational Science. Taylor & Francis, 2011. [cited at p. 26]
  - [39] A. Buluç and J. R. Gilbert. The combinatorial BLAS: design, implementation, and applications. *International Journal of High Performance Computing Applications*, 2011. [cited at p. 2]
  - [40] K. Burckhardt, D. Szczerba, J. Brown, K. Muralidhar, and G. Székely. Fast implicit simulation of oscillatory flow in human abdominal bifurcation using a Schur complement preconditioner. *Euro-Par 2009 Parallel Processing*, pages 747–759, 2009. [cited at p. 103]
  - [41] R. Burkard, M. Dell’Amico, and S. Martello. *Assignment Problems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2009. [cited at p. 20, 21, 151]
  - [42] L. Buš and P. Tvrdik. Towards auction algorithms for large dense assignment problems. *Computational Optimization and Application*, 43(3):411–436, 2009. [cited at p. 39]
  - [43] A. Caldwell, A. Kahng, and J. Markov. Design and implementation of move-based heuristics for VLSI partitioning. *ACM Journal of Experimental Algorithms*, 5, 2000. [cited at p. 26]

- [44] V. Campos, A. Duarte, M. Gallego, F. Gortázar, R. Martí, and E. Piñana. Optsicom project. <http://www.optsicom.es/mdp/>. Accessed October 2011. [cited at p. 27]
- [45] G. Carpaneto, M. Dell’Amico, and P. Toth. Exact solution of large-scale, asymmetric traveling salesman problems. *ACM Transactions on Mathematical Software*, 21:394–409, 1995. [cited at p. 18]
- [46] P. J. Carrington, J. Scott, and S. Wasserman. *Models and methods in social network analysis*. Cambridge Univ Pr, 2005. [cited at p. 17]
- [47] D. A. Castañón. Reverse auction algorithms for assignment problems. *AMS DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 12:407–430, 1993. [cited at p. 38]
- [48] Ü. V. Çatalyürek, C. Aykanat, and B. Uçar. On two-dimensional sparse matrix partitioning: Models, methods, and a recipe. *SIAM Journal of Scientific Computing*, 32(2):656–683, 2010. [cited at p. 26]
- [49] Ü. V. Çatalyürek, F. Dobrian, A. H. Gebremedhin, M. Halappanavar, and A. Pothen. Distributed-memory parallel algorithms for matching and coloring. In *2011 International Symposium on Parallel and Distributed Processing, Workshops and PhD Forum (IPDPSW), Workshop on Parallel Computing and Optimization (PCO’11)*, pages 1966–1975. IEEE Press, 2011. [cited at p. 20]
- [50] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007. [cited at p. 7]
- [51] S. C. Chen, D. J. Kuck, and A. H. Sameh. Practical parallel band triangular system solvers. *ACM Transactions on Mathematical Software*, 4(3):270–277, 1978. [cited at p. 63]
- [52] C.-T. Cheng, C. K. Tse, and F. C. Lau. A delay-aware data collection network structure for wireless sensor networks. *IEEE Sensors Journal*, 11(3):699–710, 2011. [cited at p. 18]
- [53] C. Chevalier and F. Pellegrini. PT-Scotch: A tool for efficient parallel graph ordering. *Parallel Computing*, 34(6–8):318–331, 2008. [cited at p. 26]
- [54] G. Cong and D. Bader. Techniques for designing efficient parallel graph algorithms for SMPs and multicore processors. *Parallel and Distributed Processing and Applications*, pages 137–147, 2007. [cited at p. 2]
- [55] D. Conte, P. Foggia, C. Sansone, and M. Vento. Thirty years of graph matching in pattern recognition. *International Journal of Pattern Recognition and Artificial Intelligence*, 18(3):265–298, 2004. [cited at p. 18, 95, 134]

- [56] D. Conte, P. Foggia, and M. Vento. Challenging complexity of maximum common subgraph detection algorithms: A performance analysis of three algorithms on a wide database of graphs. *Journal of Graph Algorithms and Applications*, 11(1):99–143, 2007. [cited at p. 96]
- [57] T. G. Crainic and M. Toulouse. Parallel meta-heuristics. In Michel Gendreau and Jean-Yves Potvin, editors, *Handbook of Metaheuristics*, International Series in Operations Research & Management Science, pages 497–541. Springer, 2010. [cited at p. 28]
- [58] CSCAPES. Graph coloring for computing derivatives. <http://www.cscapes.org/coloringpage/>. Accessed February 2012. [cited at p. 2]
- [59] F. E. Curtis, O. Schenk, and A. Wächter. An interior-point algorithm for large-scale nonlinear optimization with inexact step computations. *SIAM Journal on Scientific Computing*, 32(6):3447–3475, 2010. [cited at p. 106]
- [60] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 24th national conference*, pages 157–172, New York, NY, USA, 1969. ACM. [cited at p. 26]
- [61] A. Czygrinow. Maximum dispersion problem in dense graphs. *Operations Research Letters*, 27(5):223–227, 2000. [cited at p. 27]
- [62] T. A. Davis. *Direct methods for sparse linear systems*, volume 2. Society for Industrial Mathematics, 2006. [cited at p. 22]
- [63] T. A. Davis and Y. Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1), 2011. [cited at p. 108, 130, 142]
- [64] K. Deb. *Multi-objective optimization using evolutionary algorithms*. Wiley, 2001. [cited at p. 29]
- [65] K. D. Devine, E. G. Boman, R. T. Heaphy, R. H. Bisseling, and Ü. V. Çatalyürek. Parallel hypergraph partitioning for scientific computing. In *Proceedings IEEE International Parallel and Distributed Processing Symposium 2006*. IEEE Press, 2006. [cited at p. 152]
- [66] K. Diethelm. The limits of reproducibility in numerical simulation. *Computing in Science & Engineering*, 14(1):64–72, 2012. [cited at p. 123]
- [67] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959. [cited at p. 17]

- [68] S. Dill, R. Kumar, K. S. McCurley, S. Rajagopalan, D. Sivakumar, and A. Tomkins. Self-similarity in the web. *ACM Transactions on Internet Technology (TOIT)*, 2(3):205–223, 2002. [cited at p. 95]
- [69] J. J. Dongarra and A. H. Sameh. On some parallel banded system solvers. *Parallel Computing*, 1(3):223–235, 1984. [cited at p. 63]
- [70] D. E. Drake and S. Hougardy. Linear time local improvements for weighted matchings in graphs. In *WEA'03 Proceedings of the 2nd international conference on Experimental and efficient algorithms*, pages 107–119, 2003. [cited at p. 20]
- [71] R. Duan and S. Pettie. Approximating maximum weight matching in near-linear time. *Annual IEEE Symposium on Foundations of Computer Science*, pages 673–682, 2010. [cited at p. 20]
- [72] I. S. Duff, K. Kaya, and B. Uçar. Design, analysis, and implementation of maximum transversal algorithms. Technical Report TR/PA/10/76, CER-FACS, Toulouse, France, 2010. [cited at p. 8, 19, 20]
- [73] I. S. Duff and J. Koster. On algorithms for permuting large entries to the diagonal of a sparse matrix. *SIAM Journal on Matrix Analysis and Applications*, 22:973–996, 1999. [cited at p. 18, 21, 70, 108, 128]
- [74] D. Easley and J. Kleinberg. *Networks, Crowds, and Markets: Reasoning About a Highly Connected World*. Cambridge University Press, 2010. [cited at p. 34]
- [75] D. Ediger, K. Jiang, J. Riedy, and D. A. Bader. Massive streaming data analytics: A case study with clustering coefficients. In *IPDPS Workshops*, pages 1–8, 2010. [cited at p. 2]
- [76] H.-C. Ehrlich and M. Rarey. Maximum common subgraph isomorphism algorithms and their applications in molecular science: a review. *Wiley Interdisciplinary Reviews: Computational Molecular Science*, 1(1):68–79, 2011. [cited at p. 95]
- [77] A. E. Eiben and J. E. Smith. *Introduction to evolutionary computing*. Springer Verlag, 2003. [cited at p. 29]
- [78] D. Eichenberger. Entwicklung und Evaluierung von Heuristiken für das quadratische Rucksackproblem (in german). Master’s thesis, University of Basel, Switzerland, 2011. [cited at p. 82]
- [79] D. Emms, R. C. Wilson, and E. R. Hancock. Graph matching using the interference of continuous-time quantum walks. *Pattern Recognition*, 42(5):985–1002, 2009. [cited at p. 18]

- [80] P. Erdős and A. Rényi. *On the evolution of random graphs*. Akad. Kiadó, 1960. [cited at p. 17]
- [81] U. Feige, D. Peleg, and G. Kortsarz. The dense  $k$ -subgraph problem. *Algorithmica*, 29(3):410–421, 2001. [cited at p. 8, 27]
- [82] M. L. Fernández and G. Valiente. A graph distance metric combining maximum common subgraph and minimum common supergraph. *Pattern Recognition Letters*, 22(6-7):753–758, 2001. [cited at p. 95]
- [83] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *DAC '82: Proceedings of the 19th Conference on Design Automation*, pages 175–181, Piscataway, NJ, USA, 1982. IEEE Press. [cited at p. 25]
- [84] M. Fiedler. A property of eigenvectors of nonnegative symmetric matrices and its application to graph theory. *Czechoslovak Mathematical Journal*, 25(4):619–633, 1975. [cited at p. 27]
- [85] S. Fortunato. Community detection in graphs. *Physics Reports*, 486(3-5):75–174, 2010. [cited at p. 27]
- [86] Message Passing Interface Forum. <http://www.mpi-forum.org/>. Accessed April 2012. [cited at p. 6]
- [87] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615, 1987. [cited at p. 17, 20]
- [88] H. N. Gabow. Data structures for weighted matching and nearest common ancestors with linking. In *SODA*, pages 434–443, 1990. [cited at p. 19]
- [89] J. Gaidamour and P. Henon. A parallel direct/iterative solver based on a Schur complement approach. In *Computational Science and Engineering, 2008. CSE '08. 11th IEEE International Conference on*, pages 98–105, 2008. [cited at p. 62]
- [90] H. H. Gan, S. Pasquali, and T. Schlick. Exploring the repertoire of RNA secondary motifs using graph theory; implications for RNA design. *Nucleic acids research*, 31(11):2926–2943, 2003. [cited at p. 17]
- [91] X. Gao, B. Xiao, D. Tao, and X. Li. A survey of graph edit distance. *Pattern Analysis & Applications*, 13(1):113–129, 2010. [cited at p. 95]
- [92] A. H. Gebremedhin, F. Manne, and A. Pothen. What color is your Jacobian? Graph coloring for computing derivatives. *SIAM Review*, 47:629–705, April 2005. [cited at p. 15]



- [93] M. Gendreau and J. Y. Potvin. *Handbook of Metaheuristics*. International Series in Operations Research & Management Science. Springer, 2010. [cited at p. 28]
- [94] A. George. Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*, pages 345–363, 1973. [cited at p. 61]
- [95] A. J. George. *Computer implementation of the finite element method*. PhD thesis, Stanford University, Stanford, CA, USA, 1971. [cited at p. 26]
- [96] A. J. George and J. W. H. Liu. An implementation of a pseudoperipheral node finder. *ACM Transactions on Mathematical Software*, 5:284–295, 1979. [cited at p. 26]
- [97] L. Getoor and C. P. Diehl. Link mining: a survey. *ACM SIGKDD Explorations Newsletter*, 7(2):3–12, 2005. [cited at p. 95]
- [98] D. Gibson, R. Kumar, and A. Tomkins. Discovering large dense subgraphs in massive graphs. In *Proceedings of the 31st international conference on Very large data bases*, pages 721–732. VLDB Endowment, 2005. [cited at p. 27]
- [99] L. Giraud, A. Haidar, and Y. Saad. Sparse approximations of the Schur complement for parallel algebraic hybrid linear solvers in 3D. *Numerical Mathematics: Theory, Methods and Applications*, 3(3):276–294, 2010. [cited at p. 62]
- [100] M. Girvan and M. E. Newman. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences*, 99(12):7821–7826, 2002. [cited at p. 17]
- [101] D. Gleich. Code for network alignment. <http://www.cs.purdue.edu/homes/dgleich/codes/netalign/>. Accessed November 2011. [cited at p. 140]
- [102] T. Gohil, R. H. P. McGregor, D. Szczerba, K. Burckhardt, K. Muralidhar, and G. Székely. Simulation of oscillatory flow in an aortic bifurcation using FVM and FEM: A comparative study of implementation strategies. *International Journal for Numerical Methods in Fluids*, 66(8):1037–1067, 2011. [cited at p. 103, 121]
- [103] A. V. Goldberg, S. A. Plotkin, D. B. Shmoys, and È. Tardos. Using interior point methods for fast parallel algorithms for bipartite matching and related problems. *SIAM Journal on Computing*, 21:140–150, 1992. [cited at p. 21]
- [104] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum flow problem. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, STOC '86, pages 136–146, New York, NY, USA, 1986. [cited at p. 7, 20]

- [105] GPGPU. General-purpose computation on graphics hardware. <http://www.gpgpu.org/>. Accessed April 2012. [cited at p. 7]
- [106] D. Gregor and A. Lumsdaine. The parallel BGL: A generic library for distributed graph computations. *Parallel Object-Oriented Scientific Computing (POOSC)*, 2005. [cited at p. 2]
- [107] A. Griewank and A. Walther. *Evaluating derivatives - principles and techniques of algorithmic differentiation (second edition)*. SIAM, 2008. [cited at p. 15]
- [108] A. Gupta, S. Koric, and T. George. Sparse matrix factorization on massively parallel computers. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, number 1 in SC '09, pages 1–12, 2009. [cited at p. 8, 62, 154]
- [109] M. Hagemann and O. Schenk. Weighted matchings for preconditioning symmetric indefinite linear systems. *SIAM Journal of Scientific Computing*, 28:403–420, 2006. [cited at p. 18, 70, 90]
- [110] P. Hall. On representatives of subsets. *Journal of London Mathematical Society*, 10(1):26–30, 1935. [cited at p. 18]
- [111] P. Harish and P. J. Narayanan. Accelerating large graph algorithms on the GPU using CUDA. In *Proceedings of the 14th international conference on High performance computing*, HiPC'07, pages 197–208, Berlin, Heidelberg, 2007. Springer-Verlag. [cited at p. 7]
- [112] B. Hendrickson and J. W. Berry. Graph analysis with high-performance computing. *Computing in Science & Engineering*, 10(2):14–19, 2008. [cited at p. 2]
- [113] B. Hendrickson and T. G. Kolda. Graph partitioning models for parallel computing. *Parallel Computing*, 26:1519–1534, 1999. [cited at p. 23]
- [114] B. Hendrickson and R. Leland. An improved spectral graph partitioning algorithm for mapping parallel computations. *SIAM Journal on Scientific Computing*, 16:452–469, 1995. [cited at p. 26]
- [115] M. Hinze, R. Pinnau, M. Ulbrich, and S. Ulbrich. *Optimization with PDE constraints*, volume 23. Springer Verlag, 2009. [cited at p. 104]
- [116] J. E. Hopcroft and R. M. Karp. An  $n^{5/2}$  algorithm for maximum matching in bipartite graphs. *SIAM Journal on Computing*, 2:225–231, 1973. [cited at p. 19]
- [117] S. Hougardy and D. E. Vinkemeier. Approximating weighted matchings in parallel. *Information Processing Letters*, 99(3):119–123, 2006. [cited at p. 20]



- [118] HSL. The HSL mathematical software library. <http://www.hsl.rl.ac.uk/>. Accessed January 2012. [cited at p. 90, 91]
- [119] A. K. Jain. Data clustering: 50 years beyond k-means. *Pattern Recognition Letters*, 31(8):651–666, 2010. [cited at p. 22]
- [120] R. Jonker and A. Volgenant. A shortest augmenting path algorithm for dense and sparse linear assignment problems. *Computing*, 38(4):325–340, 1987. [cited at p. 21]
- [121] B. A. Julstrom. Greedy, genetic, and greedy genetic algorithms for the quadratic knapsack problem. In *Proceedings of the 2005 conference on Genetic and evolutionary computation, GECCO '05*, pages 607–614, New York, NY, USA, 2005. ACM. [cited at p. 29]
- [122] B. H. Junker and F. Schreiber. *Analysis of biological networks*, volume 4. Wiley Online Library, 2008. [cited at p. 17]
- [123] R. M. Karp. A survey of parallel algorithms for shared-memory machines. Technical report, Berkeley, CA, USA, 1988. [cited at p. 2]
- [124] G. Karypis. Family of graph and hypergraph partitioning software. <http://glaros.dtc.umn.edu/gkhome/views/metis>. Accessed January 2012. [cited at p. 90]
- [125] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48:96–129, 1998. [cited at p. 7, 152]
- [126] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359, 1999. [cited at p. 26]
- [127] G. Karypis and V. Kumar. Multilevel K-way hypergraph partitioning. In *Proceedings of the 36th annual ACM/IEEE Design Automation Conference, DAC '99*, pages 343–348, New York, NY, USA, 1999. ACM. [cited at p. 22, 26]
- [128] G. Karypis and V. Kumar. Parallel multilevel series K-way partitioning scheme for irregular graphs. *SIAM Review*, 41:278–300, 1999. [cited at p. 26]
- [129] J. Kepner and J. Gilbert. *Graph algorithms in the language of linear algebra*. Society for Industrial and Applied Mathematics, Philadelphia, 2011. [cited at p. 8, 13]
- [130] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49:291–307, February 1970. [cited at p. 25]

- [131] S. Khuller and B. Saha. On finding dense subgraphs. *Automata, Languages and Programming*, pages 597–608, 2009. [cited at p. 27]
- [132] B. S. Kirk, J. W. Peterson, R. H. Stogner, and G. F. Carey. libMesh: A C++ library for parallel adaptive mesh refinement/coarsening simulations. *Engineering with Computers*, 22(3–4):237–254, 2006. [cited at p. 22]
- [133] P. Klemperer. *Auctions: Theory and Practice*. Princeton University Press, 2004. [cited at p. 34]
- [134] G. Kollias, S. Mohammadi, and A. Grama. Network similarity decomposition (NSD): A fast and scalable approach to network alignment. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 2011 (to appear). [cited at p. 97]
- [135] G. Kollias, M. Sathe, S. Mohammadi, and A. Grama. A fast approach for the global alignment of protein-protein interaction networks. Technical report, Purdue University, 2012. submitted. [cited at p. 138]
- [136] G. Kollias, M. Sathe, O. Schenk, and A. Grama. A parallel scalable approach for graph similarity. Technical report, Purdue University, 2012. [cited at p. 142]
- [137] D. König. Graphok és alkalmazásuk a determinánsok és a halmazok elméletére [hungarian]. *Mathematikai es Természettudományi Ertesítő*, 34:104–119, 1916. [German translation: Über Graphen und ihre Anwendung auf Determinantentheorie und Mengenlehre, *Mathematische Annalen* 77:453–465, 1916.]. [cited at p. 18]
- [138] A. Kopser and D. Vollrath. Overview of the next generation Cray XMT. In *Cray User Group 2011 Proceedings*, pages 1–10, 2011. [cited at p. 153]
- [139] G. Kortsarz and D. Peleg. On choosing a dense subgraph. In *34th Annual Symposium on Foundations of Computer Science*, 1993, pages 692–701, 1993. [cited at p. 27]
- [140] V. E. Krebs. Mapping networks of terrorist cells. *Connections*, 24(3):43–52, 2002. [cited at p. 17]
- [141] O. Kuchaiev and N. Pržulj. Integrative network alignment reveals large regions of global network similarity in yeast and human. *Bioinformatics*, 27(10):1390–1396, 2011. [cited at p. 18]
- [142] H. W. Kuhn. The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2:83–97, 1955. [cited at p. 7, 19, 20]

- [143] T. Kulikova, P. Aldebert, N. Althorpe, W. Baker, K. Bates, P. Browne, A. Van Den Broek, G. Cochrane, K. Duggan, R. Eberhardt, et al. The EMBL nucleotide sequence database. *Nucleic Acids Research*, 32(suppl 1):D27–D30, 2004. [cited at p. 17]
- [144] G. Kumfert and A. Pothen. Two improved algorithms for envelope and wavefront reduction. *BIT Numerical Mathematics*, 37:559–590, 1997. [cited at p. 27, 72]
- [145] Sandia National Laboratories. Mtgl. <https://software.sandia.gov/trac/mtgl>. Accessed May 2012. [cited at p. 153]
- [146] J. Langguth. *Sequential and Parallel Matching Algorithms in Combinatorial Scientific Computing*. PhD thesis, University of Bergen, Norway, 2011. [cited at p. 20]
- [147] J. Langguth, F. Manne, and P. Sanders. Heuristic initialization for bipartite matching problems. *Journal of Experimental Algorithmics*, 15:1–22, 2010. [cited at p. 19, 38]
- [148] D. H. Lawrie and A. H. Sameh. The computation and communication complexity of a parallel banded system solver. *ACM Transactions on Mathematical Software*, 10(2):185–195, 1984. [cited at p. 63]
- [149] X. Li, M. Wu, C. K. Kwoh, and S. K. Ng. Computational approaches for detecting protein complexes from protein interaction networks: a survey. *BMC genomics*, 11(Suppl 1):S3, 2010. [cited at p. 101]
- [150] X. S. Li and J. W. Demmel. Making sparse gaussian elimination scalable by static pivoting. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, Supercomputing '98, pages 1–17, Washington, DC, USA, 1998. IEEE Computer Society. [cited at p. 62, 154]
- [151] L. Lovasz and M. D. Plummer. *Matching Theory*. AMS Chelsea Publishing, Providence, Rhode Island, 2009. [cited at p. 18, 19]
- [152] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004. [cited at p. 18, 95, 134, 135]
- [153] M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM J. Comput.*, 15(4):1036–1053, 1986. [cited at p. 16]
- [154] K. Madduri. HPC graph analysis. <http://www.graphanalysis.org/>. Accessed April 2012. [cited at p. 4]

- [155] K. Madduri. *A high-performance framework for analyzing massive complex networks*. PhD thesis, Georgia Institute of Technology, USA, 2008. [cited at p. 1]
- [156] K. Madduri, D. A. Bader, J. W. Berry, and J. R. Crobak. Parallel shortest path algorithms for solving large-scale instances. In C. Demetrescu, A. V. Goldberg, and D. Johnson, editors, *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74, pages 249–290. American Mathematical Society, 2009. [cited at p. 1, 17]
- [157] K. Mahmood, A. S. Konagurthu, J. Song, A. M. Buckle, G. I. Webb, and J. C. Whisstock. EGM: Encapsulated gene-by-gene matching to identify gene orthologs and homologous segments in genomes. *Bioinformatics*, pages 2076–2084, 2010. [cited at p. 18]
- [158] M. Manguoglu. TraceMin-Fiedler. <http://www.cs.purdue.edu/homes/mmanguog/fiedler.html>. Accessed January 2012. [cited at p. 90]
- [159] M. Manguoglu. A domain-decomposing parallel sparse linear system solver. *Journal of Computational and Applied Mathematics*, 236(3):319–325, 2011. [cited at p. 108]
- [160] M. Manguoglu, E. Cox, F. Saied, and A. H. Sameh. TraceMin-Fiedler: A parallel algorithm for computing the Fiedler vector. In *VECPAR*, pages 449–455, 2010. [cited at p. 27, 71]
- [161] M. Manguoglu, F. Saied, A. H. Sameh, and A. Grama. Performance models for the spike banded linear system solver. *Scientific Programming*, 19(1):13–25, 2011. [cited at p. 66]
- [162] M. Manguoglu, A. H. Sameh, and O. Schenk. PSPIKE: A parallel hybrid sparse linear system solver. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09, pages 797–808, Berlin, Heidelberg, 2009. Springer-Verlag. [cited at p. 8, 62, 63]
- [163] F. Manne and R. H. Bisseling. A parallel approximation algorithm for the weighted maximum matching problem. In R. Wyrzykowski, K. Karczewski, J. Dongarra, and J. Wasniewski, editors, *Proceedings Seventh International Conference on Parallel Processing and Applied Mathematics (PPAM 2007)*, volume 4967 of *Lecture Notes in Computer Science*, pages 708–717, 2008. [cited at p. 20, 130]
- [164] R. Marti, M. Gallego, A. Duarte, and E. G. Pardo. Heuristics and meta-heuristics for the maximum diversity problem. *Journal of Heuristics*, pages 1–25, 2011. [cited at p. 29]

- [165] H. Maurer and H. D. Mittelman. Optimization techniques for solving elliptic control problems with control and state constraints. part 1: Boundary control. *Comp. Optim. Applic.*, 16:29–55, 2000. [cited at p. 124]
- [166] H. Maurer and H. D. Mittelman. Optimization techniques for solving elliptic control problems with control and state constraints. part 2: Distributed control. *Computational Optimization and Applications*, 18(2):141–160, 2001. [cited at p. 124]
- [167] N. McKeown. The iSLIP scheduling algorithm for input-queued switches. *IEEE/ACM Transactions on Networking*, 7(2):188–201, 1999. [cited at p. 18]
- [168] D. Merrill, M. Garland, and A. Grimshaw. Scalable GPU graph traversal. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPOPP’12, pages 117–128, New York, NY, USA, 2012. [cited at p. 7]
- [169] U. Meyer and P. Sanders.  $\delta$ -stepping: a parallelizable shortest path algorithm. *Journal of Algorithms*, 49(1):114 – 152, 2003. [cited at p. 1, 17]
- [170] C. C. K. Mikkelsen and M. Manguoglu. Analysis of the truncated SPIKE algorithm. *SIAM Journal on Matrix Analysis and Applications*, 30(4):1500–1519, 2008. [cited at p. 63]
- [171] T. Milenković, W. L. Ng, W. Hayes, and N. Pržulj. Optimal network alignment with graphlet degree vectors. *Cancer Informatics*, 9:121–137, 2010. [cited at p. 138, 139]
- [172] F. Müller. *Remembering in the Metaverse: Preservation, Evaluation, and Context*. PhD thesis, University of Basel, Switzerland, 2011. [cited at p. 1]
- [173] J. Munkres. Algorithms for the assignment and transportation problems. *Journal of the Society for Industrial and Applied Mathematics*, 5(1):32–38, 1957. [cited at p. 7, 19, 20]
- [174] U. Naumann and O. Schenk. *Combinatorial Scientific Computing*. Chapman-Hall/CRC Computational Science. Taylor & Francis, 2011. [cited at p. 4, 15]
- [175] M. Naumov and A. H. Sameh. A tearing-based hybrid parallel banded linear system solver. *Journal of Computational Applied Mathematics*, 226:306–318, April 2009. [cited at p. 63]
- [176] Intel Software Network. Adaptive spike-based solver. <http://software.intel.com/en-us/articles/intel-adaptive-spike-based-solver/>. Accessed May 2012. [cited at p. 65, 154]

- [177] Intel Software Network. Intel math kernel library (mkl). <http://software.intel.com/en-us/articles/intel-mkl/>. Accessed January 2012. [cited at p. 88]
- [178] M. Olschowka and A. Neumaier. A new pivoting strategy for Gaussian elimination. *Linear Algebra and Its Application*, 240:131–151, 1996. [cited at p. 50, 71]
- [179] SIAM Workshop on Combinatorial Scientific Computing. <http://www.siam.org/meetings/csc11/>. Accessed October 2011. [cited at p. 4]
- [180] OpenMP. <http://www.openmp.org/>. Accessed April 2012. [cited at p. 6]
- [181] LAPACK Linear Algebra PACKage. <http://www.netlib.org/lapack/>. Accessed November 2011. [cited at p. 66, 67, 88]
- [182] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. 1999. [cited at p. 96]
- [183] P. Papadimitriou, A. Dasdan, and H. Garcia-Molina. Web graph similarity for anomaly detection. *Journal on Internet Services and Applications*, 1(1):19–30, 2010. [cited at p. 95, 142]
- [184] M. A. Patwary, R. H. Bisseling, and F. Manne. Parallel greedy graph matching using an edge partitioning approach. In *Proceedings of the Fourth ACM SIGPLAN Workshop on High-level Parallel Programming and Applications (HLPP 2010)*, pages 45–54, 2010. [cited at p. 19, 20]
- [185] T. J. Pedley. Mathematical modelling of arterial fluid dynamics. *Journal of engineering mathematics*, 47(3):419–444, 2003. [cited at p. 103]
- [186] M. Pelillo. Replicator equations, maximal cliques, and graph isomorphism. *Neural Computation*, 11(8):1933–1955, 1999. [cited at p. 95]
- [187] F. Pellegrini. Scotch and PT-Scotch graph partitioning software: An overview. In U. Naumann and O. Schenk, editors, *Combinatorial Scientific Computing*, Chapman-Hall/CRC Computational Science. Taylor & Francis, 2011. [cited at p. 26]
- [188] S. Pettie and P. Sanders. A simpler linear time  $2/3-\epsilon$  approximation for maximum weight matching. *Information Processing Letters*, 91(6):271–276, 2004. [cited at p. 20]
- [189] P. A. Pevzner. Multiple alignment, communication cost, and graph matching. *SIAM Journal on Applied Mathematics*, pages 1763–1779, 1992. [cited at p. 153]



- [190] D. Pisinger. The quadratic knapsack problem—a survey. *Discrete Applied Mathematics*, 155(5):623–648, 2007. [cited at p. 27]
- [191] E. Polizzi and A. H. Sameh. A parallel hybrid banded system solver: the SPIKE algorithm. *Parallel Computing*, 32(2):177–194, 2006. [cited at p. 63, 154]
- [192] E. Polizzi and A. H. Sameh. SPIKE: A parallel environment for solving banded linear systems. *Computers & Fluids*, 36(1):113–120, 2007. [cited at p. 63, 65]
- [193] R. Preis. Linear time  $1/2$ -approximation algorithm for maximum weighted matching in general graphs. In C. Meinel and S. Tison, editors, *STACS’99 Proceedings of the 16th Annual Conference on Theoretical Aspects of Computer Science*, volume 1563 of *Lecture Notes in Computer Science*, pages 259–269. Springer-Verlag, New York, 1999. [cited at p. 19, 20]
- [194] S. Rajamanickam, E. Boman, and M. Heroux. A hybrid solver for general sparse linear systems. Presentation, In Combinatorial Scientific Computing Workshop 2011. [cited at p. 62]
- [195] J. W. Raymond and P. Willett. Maximum common subgraph isomorphism algorithms for the matching of chemical structures. *Journal of Computer-Aided Molecular Design*, 16(7):521–533, 2002. [cited at p. 95]
- [196] J. Riedy. *Making static pivoting scalable and dependable*. PhD thesis, EECS Department, University of California, Berkeley, 2010. [cited at p. 39]
- [197] J. H. Rutgers, P. T. Wolkotte, P. K. Hölzenspies, J. Kuper, and G. J. Smit. An approximate maximum common subgraph algorithm for large digital circuits. In *Digital System Design: Architectures, Methods and Tools (DSD), 2010 13th Euromicro Conference on*, pages 699–705, 2010. [cited at p. 95]
- [198] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2003. [cited at p. 62]
- [199] Y. Saad and M. H. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 7(3):856–869, 1986. [cited at p. 62, 154]
- [200] A. H. Sameh and D. J. Kuck. On stable parallel linear system solvers. *J. ACM*, 25(1):81–91, 1978. [cited at p. 63]
- [201] A. H. Sameh and V. Sarin. Hybrid parallel linear system solvers. *International Journal of Computational Fluid Dynamics*, 12:213–223, 1999. [cited at p. 63]

- [202] M. Sathe, G. Rudolph, and K. Deb. Design and validation of a hybrid interactive reference point method for multi-objective optimization. In *IEEE World Congress on Evolutionary Computation 2008*, pages 2909–2916, 2008. [cited at p. 28]
- [203] M. Sathe, O. Schenk, and H. Burkhardt. Solving bi-objective many-constraint bin packing problems in automobile sheet metal forming processes. In *Proceedings of the 5th International Conference on Evolutionary Multi-Criterion Optimization, EMO '09*, pages 246–260, Berlin, Heidelberg, 2009. Springer-Verlag. [cited at p. 28]
- [204] M. Sathe, O. Schenk, and H. Burkhardt. An auction-based weighted matching implementation on massively parallel architectures. *Parallel Computing*, 2012. accepted for publication. [cited at p. 44]
- [205] M. Sathe, O. Schenk, M. Christen, and H. Burkhardt. A parallel PDE-constrained optimization framework for biomedical hyperthermia treatment planning. In *PARS-Mitteilungen, 22. PARS-Workshop*, 2009. [cited at p. 66]
- [206] M. Sathe, B. Uçar, O. Schenk, and A. Sameh. Towards a scalable hybrid linear solver based on combinatorial algorithms. In U. Naumann and O. Schenk, editors, *Combinatorial Scientific Computing*, Chapman-Hall/CRC Computational Science, pages 96–127. Taylor & Francis, 2012. [cited at p. 27, 66, 75, 128]
- [207] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey. Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort. In *Proceedings of the 2010 international conference on Management of data*, pages 351–362. ACM, 2010. [cited at p. 153]
- [208] O. Schenk, M. Bollhöfer, and R.A. Römer. On large-scale diagonalization techniques for the Anderson model of localization. *SIAM Journal on Scientific Computing*, 28(3):963–983, 2006. [cited at p. 8, 22, 62]
- [209] O. Schenk and K. Gärtner. PARDISO solver project. <http://www.pardiso-project.org/>. Accessed January 2012. [cited at p. 88]
- [210] O. Schenk and K. Gärtner. On fast factorization pivoting methods for symmetric indefinite systems. *Electronic Transactions on Numerical Analysis*, 23(1):158–179, 2006. [cited at p. 62, 128]
- [211] O. Schenk, A. Wächter, and M. Weiser. Inertia-revealing preconditioning for large-scale nonconvex constrained optimization. *SIAM Journal on Scientific Computing*, 31(2):939–960, 2008. [cited at p. 124]



- [212] G. Schubert, H. Fehske, G. Hager, and G. Wellein. Hybrid-parallel sparse matrix-vector multiplication with explicit communication overlap on current multicore-based systems. *Parallel Processing Letters*, 21(3):339–358, 2011. [cited at p. 67]
- [213] C. Scott and R. Nowak. Robust contour matching via the order-preserving assignment problem. *IEEE Transactions on Image Processing*, 15(7):1831–1838, 2006. [cited at p. 18]
- [214] J. Scott. Multilevel hybrid spectral element ordering algorithms. *Communications in Numerical Methods in Engineering*, 21:233–245, 2005. [cited at p. 27, 71, 72, 108]
- [215] R. Sharan, S. Suthram, R. M. Kelley, T. Kuhn, S. McCuine, P. Uetz, T. Sittler, R. M. Karp, and T. Ideker. Conserved patterns of protein interaction in multiple species. *Proceedings of the National Academy of Sciences of the United States of America*, 102(6):1974, 2005. [cited at p. 101]
- [216] T. W. Sheu and S. F. Tsai. Flow topology in a steady three-dimensional lid-driven cavity. *Computers & fluids*, 31(8):911–934, 2002. [cited at p. 8, 103]
- [217] H. D. Simon and S.-H. Teng. How good is recursive bisection? *SIAM Journal on Scientific Computing*, 18(5):1436–1445, 1997. [cited at p. 25]
- [218] R. Singh, J. Xu, and B. Berger. Pairwise global alignment of protein interaction networks by matching neighborhood topology. In *Research in Computational Molecular Biology*, pages 16–31. Springer, 2007. [cited at p. 18]
- [219] R. Singh, J. Xu, and B. Berger. Global alignment of multiple protein interaction networks with application to functional orthology detection. *Proceedings of the National Academy of Sciences*, 105(35):12763–12768, 2008. [cited at p. 98, 138, 139, 140, 141]
- [220] S. W. Sloan. An algorithm for profile and wavefront reduction of sparse matrices. *International Journal for Numerical Methods in Engineering*, 23(2):239–251, 1986. [cited at p. 26]
- [221] N. Snavely, S. M. Seitz, and R. Szeliski. Photo tourism: Exploring photo collections in 3D. In *ACM Transactions on Graphics*, pages 835–846. Press, 2006. [cited at p. 137]
- [222] A. Suzuki and T. Tokuyama. Dense subgraph problem revisited. In *Joint Workshop New Horizons in Computing and Statistical Mechanical Approach to Probabilistic Information Processing*, 2005. [cited at p. 29]

- [223] T. Suzumura, K. Ueno, H. Sato, K. Fujisawa, and S. Matsuoka. Performance characteristics of Graph500 on large-scale distributed environment. In *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, pages 149–158. IEEE, 2011. [cited at p. 153]
- [224] W. R. Taylor. Protein structure comparison using bipartite graph matching and its application to protein structure classification. In *Molecular and Cellular Proteomics*, pages 334–339, 2002. [cited at p. 18]
- [225] The SC Conference Series. The international conference for high performance computing, networking, storage, and analysis. <http://supercomputing.org/>. Accessed January 2012. [cited at p. 2]
- [226] Top500. Supercomputing sites. <http://top500.org/>. Accessed December 2011. [cited at p. 5]
- [227] A. Trifunovic and W. J. Knottenbelt. Parallel multilevel algorithms for hypergraph partitioning. *J. Parallel Distrib. Comput.*, 68(5):563–581, 2008. [cited at p. 26]
- [228] H. A. van der Vorst. Bi-cgstab: a fast and smoothly converging variant of bi-cg for the solution of nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 13(2):631–644, 1992. [cited at p. 62]
- [229] B. Vastenhouw and R. H. Bisseling. A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. *SIAM Review*, 47(1):67–95, 2005. [cited at p. 7, 22, 26, 152]
- [230] D. E. Vinkemeier and S. Hougardy. A linear-time approximation algorithm for weighted matchings in graphs. *ACM Transactions on Algorithms*, 1:107–122, 2005. [cited at p. 20]
- [231] T. von Landesberger, A. Kuijper, T. Schreck, J. Kohlhammer, J.J. van Wijk, J.-D. Fekete, and D. W. Fellner. Visual analysis of large graphs: State-of-the-art and future research challenges. *Computer Graphics Forum*, 30(6):1719–1749, 2011. [cited at p. 153]
- [232] A. Wächter and L. T. Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical Programming*, 106(1):25–57, 2006. [cited at p. 105]
- [233] S. Wagner, M. Sathe, and O. Schenk. Optimization for process plans in sheet metal forming. Technical report, 2012. submitted. [cited at p. 28]

- [234] C. Walshaw and M. Cross. JOSTLE: Parallel Multilevel Graph-Partitioning Software – An Overview. In F. Magoules, editor, *Mesh Partitioning Techniques and Domain Decomposition Techniques*, pages 27–58. Civil-Comp Ltd., 2007. [cited at p. 26]
- [235] BSP Worldwide. Co-ordinating bulk synchronous parallel computation. <http://www.bsp-worldwide.org/>. Accessed April 2012. [cited at p. 2]
- [236] X. Wu, S. Dumitrescu, P. Biyani, and Q. Wu. Fast chromosome karyotyping by auction algorithm. *International Journal of Bioinformatics Research and Applications*, 1:351–362, September 2005. [cited at p. 18]
- [237] I. Xenarios, L. Salwinski, X. J. Duan, P. Higney, S. M. Kim, and D. Eisenberg. DIP, the database of interacting proteins: a research tool for studying cellular networks of protein interactions. *Nucleic acids research*, 30(1):303–305, 2002. [cited at p. 17]
- [238] I. Yamazaki and X. S. Li. On techniques to improve robustness and scalability of a parallel hybrid linear solver. In *VECPAR’10*, pages 421–434, 2010. [cited at p. 62]
- [239] L. A. Zager and G. C. Verghese. Graph similarity scoring and matching. *Applied mathematics letters*, 21(1):86–94, 2008. [cited at p. 96]
- [240] M. Zaslavskiy, F. Bach, and J.-P. Vert. Global alignment of protein-protein interaction networks by graph matching methods. *Bioinformatics*, 25(12):1259–1267, 2009. [cited at p. 18]
- [241] Z. Zeng, A. K Tung, J. Wang, J. Feng, and L. Zhou. Comparing stars: On approximating graph edit distance. *Proceedings of the VLDB Endowment*, 2(1):25–36, 2009. [cited at p. 96]



# Appendices



# Appendix A

---

## User Manuals

---

### A.1 PSPIKE

In Fortran, the hybrid solver PSPIKE can be invoked by the following routine:

```
subroutine pspike(job, n, m, ia, ja, a, rhs, sol,
                 bandwidth, tol, nrhs, info)

integer (in) :: job(1), bandwidth(1), nrhs(1), info(2)
integer (in) :: n(1), m(1), ia(n+1), ja(m)
double precision (in) :: a(m)
double precision (inout) :: rhs(n), sol(n)
double precision (in) :: tol(1)
```

In C, PSPIKE can be invoked by the following command (note, the attached “\_” to the pspike call):

```
pspike_(int *job,
        int *n, int *m, int *ia, int *ja, double *a,
        double *rhs, double *sol,
        int *bandwidth, double *tol,
        int *nrhs, int *info)
```

#### A.1.1 Arguments of PSPIKE

Each of the arguments in the call of PSPIKE is explained in detail:

★ job — integer (1)

The solving of a linear equation system with PSPIKE is split into five phases: initialization (job = 0), preprocessing (job = 1), numerical hybrid factorization (job = 2), solving the linear equation system (job = 3), and destroy (job = 4). Thus, the flag job controls the flow over the five phases. Typically, PSPIKE need to be initialized and destroyed once in a calling program. The preprocessing phase with job = 1 and the numerical factorization with job = 2 must be called at least one time for the entire process. The actual solving of the linear system and reading of the right-hand side(s) are performed in the solving phase with job = 3.

★ n — integer (1)

The number of equations of the linear equation system is given by n,  $n > 0$ .

★ m — integer (1)

The number of nonzeros in the coefficient matrix is given by m,  $m > 0$ .

★ ia — integer (n+1)

The size of the integer array is n+1, where ia(*i*) points to the first column index of row *i* in the compressed sparse row storage format.

★ ja — integer (m)

The size of the integer array is equal to the number of nonzeros m, where ja stores the column indices in the CSR format. The indices in each row must be sorted in an increasing order.

★ a — double precision (m)

The numerical values of the matrix are stored in a (in the same order as ja) following the CSR storage format.

★ nrhs — integer (1)

Number of right-hand sides of the linear equation system.

★ rhs — double precision (n, nrhs)



The right-hand side vector or matrix. The solution overwrites this array per default.

★ `sol` — double precision ( $n$ ,  $nrhs$ )

A starting solution for the iterative solver can be provided here.

★ `bandwidth` — integer (1)

The bandwidth in relation to the upper triangular part of the matrix.

★ `tol` — double precision (1)

The stopping tolerance of the iterative solver.

★ `info` — integer (2)

The `info` array contains, on the one hand, the convergence status of PSPIKE and, on the other hand, extra options like the input of an existing preconditioner or starting solution for the iterative solver. In detail, the `info` array can be described by following Table A.1.

### A.1.2 The Option File `pspike.opt`

The user has control over the crucial aspects in PSPIKE through the option file `pspike.opt` as, for instance, the applied reordering routines or iterative/direct solver parameters. This file will be read once at the initialization phase, and mainly influences the behavior of PSPIKE. If no option file is available, PSPIKE runs in its default mode. It is recommended that one figure out the best reordering strategy for the application. In the subsequent Table A.2, the option itself, its possible values, and the description of the option are listed. The term in the column “Value” is always greater than or equal to 0. If the value is not explicitly given, then  $\mathbb{N}$  indicates an integer,  $\mathbb{R}$  a double precision number in scientific notation,  $\mathbb{B}$  a boolean with the values  $\{0, 1\}$ , and  $\mathbb{T}$  a string with at least one character. In the case of  $\mathbb{B}$ , a 1 enables the option and a 0 disables the option. The options then can be configured are given in Table A.2.

**Table A.1:** *Description of the options given to the user by the info array.*

Info Index	Description
info (1) = 0	On return, PSPIKE converges to the desired accuracy.
info (1) = -1	On return, PSPIKE was not able to converge to the desired residual. An error message is printed on the screen.
info (1) = 9	A preconditioner is given to PSPIKE in the phases before the solving phase; in the solving phase, the info flag is read and it is assumed that the linear system of interest is provided as input in this phase.
info (1) = -2	The partitioner/spectral heuristic is called once for the entire process (only of interest for the preprocessing phase)
info (2) = -1	The option file pspike.opt2 should be read.
info (2) = -2	A starting solution is given to the iterative solver (only of interest for the solving phase). Per default the starting solution is filled with zeros in the iterative solver.

**Table A.2:** *Description of the configuration file pspike.opt.*

Name	Value	Description
output_file	T	Name of the debug output file. The MPI rank is appended on the end of the string automatically.
standalone_version	B	If enabled PSPIKE has the full control over the termination process (see also option with_ipopt).
is_matrix_symmetric	B	If enabled on entry, only the upper triangular part of the matrix is stored.
pspike_tol	R	Desired tolerance of the relative residual in PSPIKE; only enabled if the parameter pspike_tol in the call of the interface is zero on entry.
bicgstab_max_iter	N	Number of the maximum BICGSTAB iterations.
bicgstab_inner_tol	R	Desired tolerance of the relative residual in the inner BICGSTAB solver.
bicgstab_inner_iter	N	Number of maximum iterations for the inner BICGSTAB solver.

**Table A.2:** *Description of the configuration file pspike.opt.*

Name	Value	Description
with_prec_bicgstab_inner	B	If enabled, the preconditioner in the inner BICGSTAB is used.
bicgstab_max_stagnation	N	If the relative residual in the outer BICGSTAB solver does not improve within the given number, then the current best solution is returned.
pspike_bandwidth	N	The bandwidth $> 0$ of the PSPIKE solver can be controlled over the option file if the parameter bandwidth in the call of the interface is zero on entry.
with_fast_matvec	0	In the SpMV in BICGSTAB, the entire vector is gathered across the processes.
	1	In the SpMV in BICGSTAB, entries of the vector are only communicated to a process if required by the corresponding process.
pspike_debug_level	0 – 6	Debug print level of PSPIKE; messages are printed in the debug output file for each process.
pardiso_iparm_31	0 – 2	Control over PARDISO option “Partial solve for sparse right-hand sides and sparse solution.”
pardiso_32bit	B	If enabled, the 32-bit factorization (see iparm(29)) of PARDISO is used.
pardiso_msglvl	N	If enabled, PARDISO prints debug messages on stdout.
pardiso_iter_ref	N	Number of iterative refinement steps in PARDISO
with_scaling	B	Enables the scaling of the entire matrix in general.
scaling_symmetric	B	Enables the $2 \times 2$ matching including its scaling.
with_mc64_scaling	B	Enables the MC64 scaling method.
with_auction_scaling	B	Enables the auction scaling method.
with_rcm_reordering	B	If enabled, RCM is used.
with_mc64_matching	B	Enables the MC64 matching implementation.

**Table A.2:** *Description of the configuration file pspike.opt.*

Name	Value	Description
with_mc73_spectral	$\mathbb{B}$	Enables the spectral heuristic implementation MC73.
with_mc73_hager	$\mathbb{B}$	If enabled, Hager method is enabled in the implementation MC73.
with_tracemin_fiedler	$\mathbb{B}$	Enables the spectral heuristic TraceMin-Fiedler.
fiedler_tolerance	$\mathbb{R}$	Desired residual for the Fiedler vector computed in spectral heuristics.
with_fill_couplings	$\mathbb{B}$	Enables the reordering of the weighted subgraph problem in general.
with_sort_heuristic	0	DEMIN with objective to maximize the weight in the subgraph is used.
	1	FIRSTFIT with MAXENTRY is used.
	2	DEMIN with objective to maximize the number of nonzeros in the subgraph is used.
	3	The $(1 + 1)$ -EA is used.
with_metis	$\mathbb{B}$	If enabled, partitioner Metis is used.
with_auction	$\mathbb{B}$	Enables the auction algorithm.
with_mondriaan	$\mathbb{B}$	If enabled, partitioner Mondriaan is used.
with_ml_sloan	$\mathbb{B}$	Enables the multilevel Sloan algorithm (implemented in MC73) as a postprocessing routine.
with_ipopt	$\mathbb{B}$	If enabled, IPOPT controls the quality of the solution. Interaction with IPOPT is enabled to find an acceptable solution to do the next inexact optimization step.
ipopt_inexact_iter	$\mathbb{N}$	After this number of BICGSTAB iterations the interaction starts with IPOPT.
ipopt_inexact_tol	$\mathbb{R}$	At this tolerance in PSPIKE the interaction starts with IPOPT.
print_matrix	$\mathbb{B}$	Enables printing of matrices in general.
print_matrix_csr	$\mathbb{B}$	Print all matrices in the CSR format.
print_matrix_mtx	$\mathbb{B}$	Print all matrices in the Matrix Market format.
print_matrix_matlab	$\mathbb{B}$	Print all matrices suitable for Matlab.

### A.1.3 A Small Example

The linear equation system  $\mathcal{A}\mathbf{x} = \mathbf{f}$  is solved (cf. Fig. 2.1) where

$$\mathcal{A} = \begin{pmatrix} 9 & 6 & 0 & 3 & 0 & 2 \\ 0 & 2 & 7 & 0 & 1 & 0 \\ 5 & 4 & 0 & 0 & 0 & 3 \\ 0 & 6 & 8 & 3 & 4 & 0 \\ 8 & 0 & 4 & 0 & 1 & 0 \\ 0 & 0 & 0 & 7 & 6 & 5 \end{pmatrix}, \mathbf{b} = \begin{pmatrix} 20 \\ 10 \\ 12 \\ 21 \\ 13 \\ 18 \end{pmatrix} \text{ with the solution } \mathbf{x} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}.$$

The test program can be compiled using the GNU compiler suites with the following command

```
mpif90 -O3 -o TestPspike TestPspike.f90 libpspike.so lib-
pardiso.so liblapack.so libblas.so -fopenmp
```

```

1 PROGRAM main
2
3     implicit none
4
5     include 'mpif.h'
6
7     ! Function parameter declaration
8     external PSPIKE
9
10    ! Linear equation system including an example
11    ! matrix stored in 1-based CSR format
12    integer :: neqns = 6
13    integer :: nnz = 20
14    integer :: ia(7)          = (/1,5,8,11,15,18,21/)
15    integer :: ja (20)        = (/1,2,4,6,2,3,5,1,2,6,
16                                2,3,4,5,1,3,5,4,5,6/)
17    double precision :: a(20) = (/9,6,3,2,2,7,1,5,4,3,
18                                6,8,3,4,8,4,1,7,6,5/)
19    ! right-hand side b and solution x
20    double precision :: b(6) = (/20,10,12,21,13,18/)
21    double precision :: x(6) = (/0,0,0,0,0,0/)
22
23    ! PSPIKE parameters
24    integer :: bandwidth
25    double precision :: bicgstab_tol
26    integer :: nrhs, info(2)
27
28    ! MPI Variables
29    integer error, nProcs, rank
30
```

```

31      ! MPI Settings
32      call MPI_INIT (error)
33      call MPI_COMM_SIZE (MPI_COMM_WORLD, nProcs, error)
34      call MPI_COMM_RANK (MPI_COMM_WORLD, rank, error)
35
36      !!! Setting the default parameter
37      bandwidth = 0
38      bicgstab_tol = 0
39      info = 0
40      nrhs = 1
41
42      !! Initialize PSPIKE (job = 0)
43      call PSPIKE(0, neqns, nnz, ia, ja, a, b, x,
44                bandwidth, bicgstab_tol, nrhs, info)
45      !! Preprocessing PSPIKE (job = 1)
46      call PSPIKE(1, neqns, nnz, ia, ja, a, b, x,
47                bandwidth, bicgstab_tol, nrhs, info)
48      !! Numerical Hybrid Factorization (job = 2)
49      call PSPIKE(2, neqns, nnz, ia, ja, a, b, x,
50                bandwidth, bicgstab_tol, nrhs, info)
51      !! Solving System (job = 3)
52      call PSPIKE(3, neqns, nnz, ia, ja, a, b, x,
53                bandwidth, bicgstab_tol, nrhs, info)
54      !! Destroy PSPIKE (job = 4)
55      call PSPIKE(4, neqns, nnz, ia, ja, a, b, x,
56                bandwidth, bicgstab_tol, nrhs, info)
57
58      call MPI_FINALIZE(error)
59
60      END PROGRAM main

```

**Listing A.1:** Example Fortran Program TestPspike.f90

The corresponding option file pspike.opt looks as follows:

```

1  output_file pspike.out
2  standalone_version 1
3  is_matrix_symmetric 0
4  matrix_stays_symmetric 0
5  bicgstab_tolerance 1e-10
6  bicgstab_max_iter 500
7  bicgstab_max_stagnation 20
8  bicgstab_inner_tol 1e-15
9  bicgstab_inner_iter 100
10 with_prec_bicgstab_inner 1
11 pspike_bandwidth 1

```

```

12 pspike_debug_level 2
13 pardiso_iparm_31 2
14 pardiso_32bit 0
15 pardiso_msglvl 0
16 pardiso_iter_ref 10
17 distribute_rowblockwise 1
18 with_fast_matvec 0
19 with_scaling 1
20 scaling_symmetric 0
21 with_matching_dualscaling 1
22 with_auction_dualscaling 0
23 with_rcm_reordering 0
24 with_mc64_matching 1
25 with_mc73_spectral 1
26 with_fill_couplings 1
27 with_sort_heuristic 3
28 withmetis_partitioner 0
29 with_auction_matching 0
30 with_tracemin_fiedler 0
31 with_mondriaan 0
32 with_ml_sloan 0
33 print_matrix_csr 0
34 print_matrix_mtx 0
35 print_matrix_matlab 0
36 print_matrix 0
37 with_ipopt 0
38 ipopt_inexact_iter 1
39 ipopt_inexact_tolerance 1e1
40 with_mc73_hager 0
41 fiedler_tolerance 1e-5

```

**Listing A.2:** *Options File pspike.opt for the TestPspike.f90*

## A.2 PAUL

PAUL computes weighted matchings in sparse and dense bipartite graphs and can be invoked from C with the following call:

```

auction_(int *ia, int *ja, double *a, int *n, int *m,
         int *perm, double *scaling,
         int *displs, int *option,
         int *nRows)

```

\* n — integer (1)

The number of equations of the local linear equation system is given by  $n$ ,  $n > 0$ .

★  $m$  — integer (1)

The number of nonzeros in the local coefficient matrix is given by  $m$ ,  $m > 0$ .

★  $ia$  — integer ( $n+1$ )

The size of the integer array is  $n+1$ , where  $ia(i)$  points to the local first column index of row  $i$  in the compressed sparse row storage format.

★  $ja$  — integer ( $m$ )

The size of the integer array is equal to the number of nonzeros  $m$ , where  $ja$  stores the column indices in the local CSR format. The indices in each row must be sorted in an increasing order.

★  $a$  — double precision ( $m$ )

The numerical values of the local matrix are stored in  $a$  (in the same order as  $ja$ ) following the CSR storage format. Note, the values are overwritten on exit.

★  $perm$  — integer ( $n$ )

The matching obtained by the auction algorithm stored as a permutation vector.

★  $scaling$  — double precision ( $2n$ )

The dual variables of the auction algorithm can be used as scaling vectors; the first  $n$  entries of  $scaling$  are the scaling factors  $u$  for the rows, and the last  $n$  entries are the scaling factors  $v$  for the columns of the matrix.

★  $displs$  — integer ( $P$ )

Starting global row-block index of the distributed matrix for each process of  $P$ .

★  $nRows$  — integer (1)



Size of the entire matrix.

★ option — double precision (13)

Options like the  $\varepsilon$ -scaling strategy can be configured, here. For details see Table A.3.

**Table A.3:** *Description of the options given to the user by the option array.*

Option Index	Value	Description
option [0]	0 – 3	Debug print level.
option [1]	$\mathbb{N}$	The value of $\zeta$ .
option [2]	$\mathbb{N}$	The value of $\theta$ .
option [3]		Unused.
option [4]	$\mathbb{N}$	Number of maximum auction iterations.
option [5]		Unused.
option [6]	$\mathbb{R}$	The minimum percentage of how many rows should be matched.
option [7]	$\mathbb{B}$	If enabled, the matrix is already distributed among processes and each process holds a 0-based block matrix stored in CSR format.
option [8]	$\mathbb{B}$	If enabled, the parameter <code>displs</code> is taken as the underlying distribution.
option [9]	$\mathbb{B}$	If enabled, the matrix is a dense matrix.
option [10]	$\mathbb{B}$	If enabled, the dual variables are gathered and returned at termination of the algorithm.
option [11]	$\mathbb{B}$	If enabled, the $\varepsilon$ -PAA with the adaptive $\varepsilon$ -scaling strategy is used.
option [12]	$\mathbb{B}$	If enabled, the numerical values of the matrix are already scaled to positive values. Otherwise an internal scaling mechanism is applied.

### A.2.1 A Small Example

For the following matrix  $\mathcal{A}$  a weighted matching is computed (cf. Fig. 2.4) where

$$\mathcal{A} = \begin{pmatrix} 9 & 6 & 0 & 3 & 0 & 2 \\ 0 & 2 & 7 & 0 & 1 & 0 \\ 5 & 4 & 0 & 0 & 0 & 3 \\ 0 & 6 & 8 & 3 & 4 & 0 \\ 8 & 0 & 4 & 0 & 1 & 0 \\ 0 & 0 & 0 & 7 & 6 & 5 \end{pmatrix}.$$

The test program can be compiled using the GNU compiler suite with the following command

```
mpicc -O3 -o TestAuction TestAuction.c libauction.so -fopenmp
```

```

1  #include "mpi.h"
2  #include "stdio.h"
3
4  int auction_(int * ia, int * ja, double * a,
5              int * neqns, int * nnz,
6              int * perm, double * scaling,
7              int * displs, double * option,
8              int * nRows);
9
10 int main(int argc, char ** argv){
11
12     // Initialize MPI
13     MPI_Init (&argc, &argv);
14
15     // Linear equation system including an example
16     // matrix stored in 0-based CSR format
17     int neqns = 6;
18     int nnz = 20;
19     int ia[] = {0, 4, 7, 10, 14, 17, 20};
20     int ja[] = {0, 1, 3, 5, 1, 2, 4, 0, 1, 5,
21               1, 2, 3, 4, 0, 2, 4, 3, 4, 5};
22     double a[] = {9, 6, 3, 2, 2, 7, 1, 5, 4, 3,
23                  6, 8, 3, 4, 8, 4, 1, 7, 6, 5};
24
25     // auction algorithms input parameter
26     int perm[neqns];
27     double scaling[12];
28     double option[13];
29     int * displs = NULL;
30
31     int i;
```

```
32     for (i = 1; i < 13; i++) option[i] = 0.0;
33
34     option[12] = 0; // values are already scaled
35     option[0] = 3; // print debug output
36
37     auction_(ia, ja, a, &neqns, &nnz, perm, scaling,
38             displs, option, &neqns);
39
40     for (i = 0; i < 6; i++){
41         printf("Buyer %d is matched
42               with object %d\n", i, perm[i]);
43     }
44
45     /*
46     print the result
47     Buyer 0 is matched with object 1
48     Buyer 1 is matched with object 2
49     Buyer 2 is matched with object 5
50     Buyer 3 is matched with object 4
51     Buyer 4 is matched with object 0
52     Buyer 5 is matched with object 3
53     */
54
55     // Finalize MPI
56     MPI_Finalize();
57 }
```

**Listing A.3:** *Example C Program TestAuction.c*



---

# List of Symbols

---

Symbol	Description
$\mathcal{A} = (a_{ij})$	matrix
$\mathcal{A}^\top$	transpose of $\mathcal{A}$
$\mathcal{A}^b$	banded matrix
$\mathcal{A}^r$	reordered matrix
$\mathcal{A}_i^r$	rectangular part $i$ of $\mathcal{A}^r$
$A_1, \dots, A_K$	square block matrices
$a$	numerical values
$\bar{a}_i$	$\max_{j=1, \dots, n}  a_{ij}  \neq 0$
$\bar{a}_{\max}$	maximum entry in $\mathcal{A}$
$\bar{a}_{\min}$	minimum entry in $\mathcal{A}$
$B_i, C_i$	coupling block matrices
$BC^\top$	matrix containing $B, C$
$B_i^c, C_i^c$	candidate block matrices
$\mathbf{b}$	right-hand side vector
$c$	cost function $c : \mathcal{V} \rightarrow \mathbb{R}$
$\text{co}_i$	cover rate of block $i$
$D$	square block diagonal matrix
$D_r, D_c$	scaling matrix for rows/columns
$\mathcal{E}$	set of edges
$\mathcal{E}_h$	set of hyperedges
$\mathcal{G}$	general graph
$\mathcal{G}_b$	bipartite graph
$\mathcal{G}_h$	hypergraph
$\mathcal{H}$	general subgraph
$\mathcal{H}_b$	bipartite subgraph
$I$	set of unassigned buyer
$ia$	array of row pointers

---

Symbol	Description
$ja$	column indices
$K$	number of partitions
$K_f$	reordering obtained by solving the weighted dense subgraph problem
$k$	bandwidth (in the context with PSPIKE)
$LU$	$LU$ decomposition
$\mathcal{M}$	matching set
$m$	number of edges
$n_1, n_2$	number of vertices in $\mathcal{V}_1, \mathcal{V}_2, n_1 \leq n_2$
$n$	$\max n_1, n_2$
$P$	maximum number of processes
$P_{s_1}, P_{s_2}$	reordering obtained by graph partitioner or spectral heuristic
$Q_m$	reordering obtained by graph matching
$R_o$	reordering obtained by Sloan's algorithm
$\mathcal{S}$	similarity matrix
$S$	spike matrix
$T$	maximum number of threads
$T_p$	time (in seconds) using $p$ processes
$T_S$	time (in seconds) using 1 process
$\mathcal{V}$	set of vertices
$\mathcal{V}_1$	vertex set of the left part of $\mathcal{G}_b$
$\mathcal{V}_2$	vertex set of the right part of $\mathcal{G}_b$
$V_i, W_i$	dense spike matrices
$w$	weight function $w : \mathcal{E} \rightarrow \mathbb{R}$
$w_{ij}$	$:= w(i, j)$
$\mathbf{x}$	solution vector
1-D	one-dimensional
2-D	two-dimensional
3-D	three-dimensional
$\Delta$	maximum vertex degree in $\mathcal{G}$
$\delta_{\text{Max Entry}}$	quality measure takes the maximum entry per row/column
$\delta_{\text{Sum}}$	quality measure adds all values in a row/column
$\delta_{\text{Sp}}$	quality measure computes the scalar product
$\star_{\text{local}}$	variable only visible to one process
$\star_{\text{global}}$	variable visible to all processes
$\varepsilon$	scaling parameter in PAUL
$\varepsilon_{\text{in}}$	tolerance for the inner BICGSTAB

Symbol	Description
$\varepsilon_{\text{out}}$	tolerance for the outer BICGSTAB
$\epsilon$	small increment
$\Pi_r, \Pi_c$	reordering of rows/columns
$\theta, \xi$	scaling parameters for the $\epsilon$ -PAA
$\star^{\mathcal{B}, \mathcal{M}, \mathcal{T}}$	bottom, middle, or top part





---

# Index

---

- (1 + 1)-EA, 82
- K*-way partition, 23
- adjacency matrix, 14
- alignment graph, 101
- aortic aneurysm, 102
- approximation matching algorithm
  - approx, 130
- auctions, 33
- augmenting path, 19
- balanced, 14
- bandwidth, 15
- benefit, 34
- BiCGStab, 62
- bipartite graph, 13
- block diagonal matrices, 63
- candidate block, 71
- compressed sparse column
  - CSC, 14
- compressed sparse row
  - CSR, 14
- conserved edges, 101
- coordinate list
  - COO, 14
- coupling block matrices, 63
- cover rate, 74
- Cray clusters
  - Cray XE6, 107
  - Cray XK6, 107
- DeMin, 79
- direct solver, 61
- evolutionary algorithms
  - EAs, 29
- Fiedler vector, 26
- FirstFit, 78
- Florida Sparse Matrix Collection, 130
- graph isomorphism, 95
- greedy matching
  - greedy, 19
- hybrid solver, 61
- hypergraph, 14
- Ipopt, 105
- isomorphism, 13
- iterative solver, 62
- matching, 13
- matching markets, 34
- MC64, 21
- MC73, 27
- Message Passing Interface
  - MPI, 6
- metaheuristics, 27
- multilevel hybrid partitioning, 72
- network similarity decomposition
  - NSD, 97
- Newton system
  - KKT system
    - saddlepoint system, 106
- OpenMP, 6
- parallel auction algorithm
  - PAA, 44
- Pardiso, 62

- partition, 13
- path, 13
- PetSc, 114
- price, 34
- profile, 15
- profit, 34
- PSPIKE, 62
  
- reverse Cuthill-McKee
  - RCM, 26
  
- similarity matrix, 97
- similarity rate, 101
- Sloan's Algorithm, 26
- sparse matrix–vector multiplication, 67
- spike matrix, 63
- subgraph, 13
  
- TOP500, 5
- TraceMin-Fiedler, 27
  
- weighted graph, 13

---

# Curriculum Vitae

---

## Personal Data

Name	Madan Sathe
Date of birth	June 03, 1981
Parents	Manohar and Smita Sathe
Nationality	German
Citizenship	Borken (Westf.), Germany

## Education

1991 – 2000	Gymnasium Marianhiller Missionare, Maria Veen (Germany)
2001 – 2007	Diploma in Computer Science, minor in Business Administration at the TU Dortmund (Germany)
2007 – 2012	Ph.D. candidate in Computer Science at the University of Basel (Switzerland)
13.08.2012	Ph.D. examination, University of Basel (Switzerland)

I enjoyed attending the lectures of the following professors and lecturers:

J. Biskup, P. Buchholz, E.-E. Doberkat, G. A. Fink, M. Grote, C. Hamilton, T. Jansen, C. Jelger, H. Holzmüller, E. Hüllermeier, G. Kern-Isberner, R. Lackes, W. Leininger, P. Marwedel, P. Padawitz, P. Recht, B. Reusch, H. J. Richter, G. Rudolph, B. Steffen, B. Vöcking, J. Wahl, H. F. Wedde, I. Wegener